JavaScript для тупых

Оглавление

[**Братки JavaScript’a** 3](#_Toc103950820)

[**Переменные JavaScript’a** 3](#_Toc103950821)

[**Переменные. Const, let, var** 4](#_Toc103950822)

[**Преобразование типов** 6](#_Toc103950823)

[**Операторы сравнения** 9](#_Toc103950824)

[**Сравнение разных типов.** 9](#_Toc103950825)

[**Типы данных. Number** 11](#_Toc103950826)

[**Тип данных String.** 12](#_Toc103950827)

[**Типы данных. null и undefined.** 13](#_Toc103950828)

[**Типы данных. Оператор typeof** 14](#_Toc103950829)

[**Условные операторы: if и ?** 15](#_Toc103950830)

[**Циклы: while, for.** 17](#_Toc103950831)

[**Операторы** 18](#_Toc103950832)

[**Знакомство со switch** 22](#_Toc103950833)

[**Функции** 24](#_Toc103950834)

[**Функции. Возврат значения.** 26](#_Toc103950835)

[**Стрелочные функции** 26](#_Toc103950836)

[**Знакомство с DOM** 28](#_Toc103950837)

[**DOM-элементы. Получение объектов.** 29](#_Toc103950838)

[**DOM-элементы. События. Ввод.** 30](#_Toc103950839)

[**DOM – элементы. События. часть 2.** 36](#_Toc103950840)

[**Объекты** 39](#_Toc103950841)

[**Массивы** 43](#_Toc103950842)

[**Методы массивов: splice, slice, concat.** 48](#_Toc103950843)

[**Методы массивов: find, findIndex, filter** 52](#_Toc103950844)

[**Методы массивов: map, sort.** 56](#_Toc103950845)

[**Методы массивов: reverse, split, join.** 58](#_Toc103950846)

[**Методы массивов: indexOf, lastIndexOf, incudes** 60](#_Toc103950847)

[**Методы массивов: reduce, spread** 61](#_Toc103950848)

[**Методы массивов: Array.isArray, some, every** 64](#_Toc103950849)

[**Методы массивов: flat, fill.** 65](#_Toc103950850)

[**Планирование вызова функции: setTimeout** 68](#_Toc103950851)

[**Планирование вызова функции: setInterval** 71](#_Toc103950852)

[**Объекты. Свойства** 72](#_Toc103950853)

[**Объекты. Методы.** 80](#_Toc103950854)

[**This, call, apply, bind** 82](#_Toc103950855)

[**Замыкания** 90](#_Toc103950856)

[**Прототипы** 93](#_Toc103950857)

[**Promiese (Обещания)** 100](#_Toc103950858)

**Братки JavaScript’a**

**Примеры парных тегов:**

**<b>/<b>** - тег жирного текста

**<u></u>** - тег подчеркнутого текста.

**Примеры не парных тегов:**

**<br>** - тег разрыва строки

**<hr>** - тег горизонтальной линии

**<img>** - тег картинки

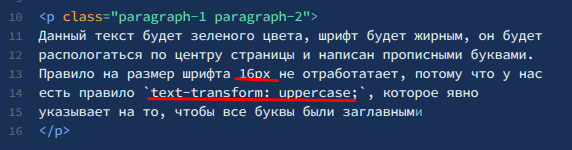
Такое css свойство как

**font-size: 16px,** будет перезаписан

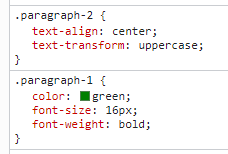
**text-transform: uppercase;**

Я не знал.

И как оказалось правильно не знал. Это ошибка в книге.

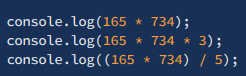


Если прописать оба класса, то ни одно из свойств не проигнорируется. Может это раньше так работало? Может в другом браузере. Не знаю.



**Переменные JavaScript’a**

Типичный JS код



В третьем выражении, кстати, пропущено умножение на 3. Раз он хотел вывести все значения.

Но смысл понятен. Зачем заставлять компьютер вычислять одно и то же заново. Можно создать для этого, ладно не перебиваю урок…

**Переменная –** это выделенная именованная область в памяти для хранения каких-либо данных.

Тот же код, что и выше. Но с использованием переменной.



Вторая и третья строка просто выводят значения операций. На значение переменной productNumbers оно никак не влияет.

В JavaScript есть 3 способа объявления переменной:

**Let, var, const.**

****

Чем различаются об этом позже.

**Переменные. Const, let, var**

Const, let, var это операторы.

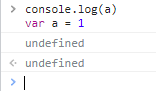
**Var – старый оператор, который встречает только в давно написанных проектах (упс).**

Он определяет переменной в текущей области видимости. Область видимости переменной – это функция внутри которой она объявлена.

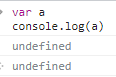
**Если переменную объявили не внутри функции, то ее область видимости - глобальный объект window.**

****

Данный код выведет undefined(не определена), т.к. функция вывода переменной “**a”** вызывается раньше, чем эта переменная была объявлена. Но хоть переменная объявлена после функции вывода, но она все же была объявлена. Поэтому вывод будет такой:

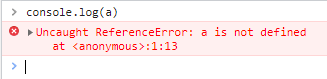


Для строчки console.log(a) переменная уже определена, просто у нее еще нет никакого значения. Тоже самое будет если написать так:





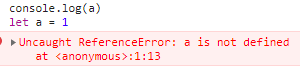
Но если попытаться просто вывести значение этой переменной не объявляя ее будет такая ошибка:



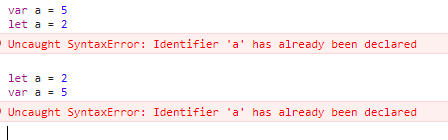
Так как переменная не определена или не объявлена

**Let –** очень похож на **var.** Единственное различие – **область видимости.**

Если прописать тот же код, но с этой переменной, то будет такая ошибка.



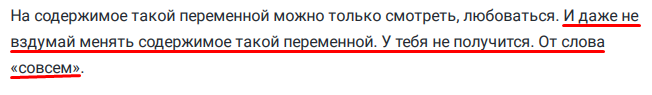
Если пытаться переопределить переменную с var на let или наоборот тоже будет ошибка.



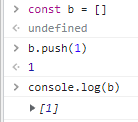
**Let** более предсказуемый. Вместо var теперь только его использовать.

**Const**

Оператор константа. Объявлять его можно только 1 раз, иначе будет ошибка. И изменять его значение тоже нельзя(Кроме объектов и массивов) При объявлении обязательно заполнить переменную каким-то значением.



Спорим?



**Преобразование типов**

Способы преобразования числа в строку:

**let ageNumber = 19;**

**let ageString = ageNumber + ‘ ‘;** // Странный вариант, старый

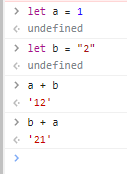
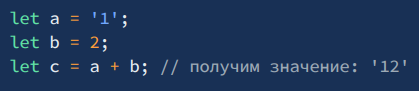
**let ageString = `${ageNumber}`** // Уже лучше, кстати кавычки в этом коде не обычные, а те,

// что на букву ё

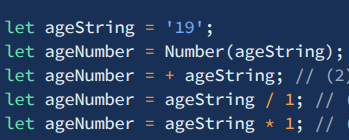
**let ageString = String(ageNumber**) // самый порядочный способ, функция String должна писаться // с заглавной буквы.

Но такое преобразование не везде применимо. Для примитивных типов данных вроде: N**umber, String, Boolean, null и undefined** это сработает. Но, такое преобразование не прокатит с **объектами.**

**Важно,** при сложении переменных, если хотя бы одна из них имеет строковый тип, то может произойти сложение типа 1 + “2” = “12”



Чтобы корректно складывать числа в таких случаях необходимо уже строку преобразовать в **Number.** С таким преобразование уже по сложнее… Вот как это можно сделать. Воспользоваться функцией, поделить или умножить строку на единицу или прописать + перед строкой.

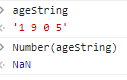


Но корректнее, конечно, будет использовать именно функцию **Number()** – явное преобразование.

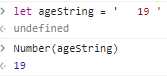
Если преобразование пройдет, то строка превратится в число или цифру, в противном случае функция вернет **Nan(Not-A-Number**) – не число.



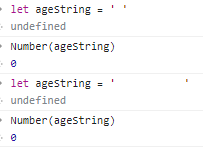
Тут имеется ввиду не пробелы между цифрами. В таких случаях будет возвращен NaN.



А именно пробелы вокруг цифр.

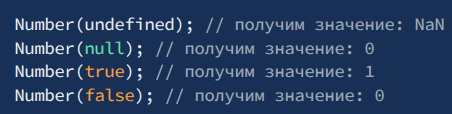


Если же строка не будет содержать никаких символов, а только пробелы, то будет возвращен ноль.



Не важно сколько пробелов, даже если их нет вообще.

Из нескольких особенностей функции **Number:**





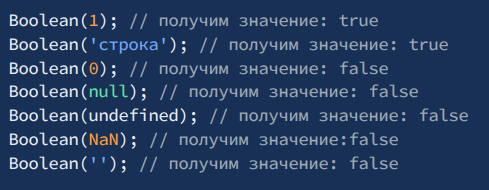
Вариант **+ ageString,** считается полезным лайфхаком. В этом случае оператор **+** полностью заменяет работу функции Number(), т.к. их поведение полностью совпадает. В принципе практика не плохая.

Два последних варианта



Так делать не стоит. Просто пример, что операции умножения и деления тоже могут осуществлять операции приведения типов.

Преобразование к **Boolean** проще всего. **true** если это не пустая строка или единица. Во всех остальных **false**. Причем если строка содержит хотя бы 1 пробел, то она уже не будет считаться пустой.

**Операторы сравнения**

Операции сравнения понятны.

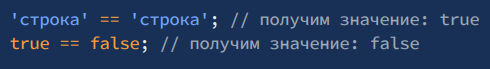


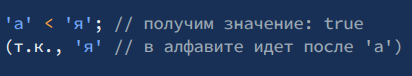
Но я слышал, что вместо **==** лучше ставить **===**

Одно **=** это, понятное дело, оператор присвоения.

Операторы сравнения возвращают результат булевого типа. То есть **true** или **false.**

Сравнивать также можно строки, символы и другие типы данных:



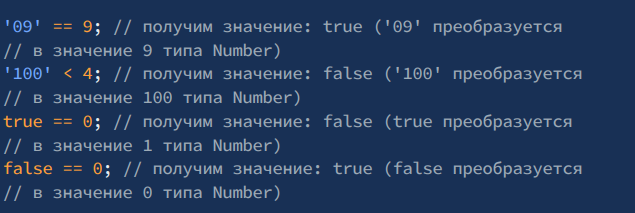


В случаи сравнения символ проверка идет по порядку символ в таблице **Unicode,** которую js использует.

**Сравнение разных типов.**



и подобное это понятно



Если сравнить строки, то сравниваться они будут посимвольно.

Таким образом такое сравнение:



эквивалентно : ‘б’ > ‘а’

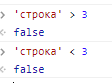
Сравнение по последним символам, т.к. остальные одинаковые. Если первых символ будет больше, то остальные уже не будут считаться.



С другой стороны, если попытаться сравнить строку с символом:

**‘строка’ > 3**

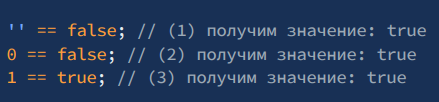
То с любым оператором все равно будет **false.**



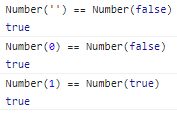
JS будет пытаться преобразовать строку в число и возвращать Nan. И результат такого сравнения всегда **false.**

**Строгое сравнение**

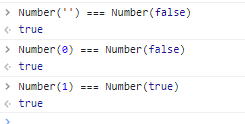
А при сравнении помимо оператора **==** можно еще использовать оператор **===** так называемое строгое сравнение. Разница от обычного есть. При обычном сравнение JS пытается привести сравниваемые значению к типу **Number.** Поэтому могут быть такие результаты.



Это тоже самое, что и



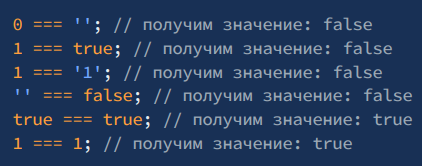
Написав тоже самое с тремя равно, ничего не изменится



Но если сделать это с первой запись, то результаты будут такими:

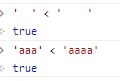


Строгое равно необходимо, чтобы отличать скажем типы. И Boolean false === только Boolean false.



Потому что, функция Boolean для не пустой строки возвращает true

Разное количество пробелов в строках, хотя тут и обычное сравнение покажет лож.

Если символы одинаковые, то решает количество

**Типы данных. Number**

В обычном JavaScript используется динамическая типизация (**TypeScript** когда-нибудь). Это значит, что типы данных в языке есть, но переменные не привязываются к ним.

Также в JavaScript есть типы данных, которые отсутствуют в других языках, о них позже.

**Number –** к этому типу относят целочисленные значения и значения с плавающей точкой.



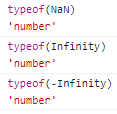
Это не совсем типично для языков программирования. Обычно такие переменные были бы разных типов данных **integer** и **float.**

Над переменными числового типа можно производить любые математические операции.

К типу **Number** относятся еще несколько **специальных** числовых значений. Это:

**Nan, Indinity, - Infinity.**

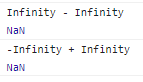
Это можно свободно проверить, с помощью функции typeof(), которая принимает значение, а возвращает тип.



Хотя забавно, Nan не число, но числовой тип. **Nan возникает если строку не удалось преобразовать в число.**



Бесконечность и минус бесконечность можно получив, поделив число на ноль или минус ноль.

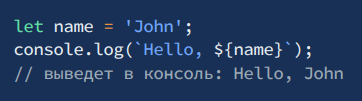
 

Из-за математических операций JS код не сломается, худшее, что может случится. Это получение значения NaN там, где планировалось получить число.

**Тип данных String.**

Строки можно заключать в двойные и одинарные кавычки. Ну нужно, в **одинарные!** Правило хорошего тона**. “Нет”, ‘Да’, `Особый случай`** - обратные кавычки нужны для работы со спец строками. Не помню, как называются. Но есть такие. Обратные кавычки появились с форматом ES6.

Пример:

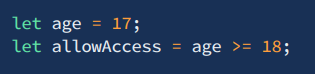


Чтобы вставить в такую строку переменную нужно использовать конструкцию: **${переменная}** внутри таких кавычек.

Внутри **${…}** можно не только вставлять переменные, но вызывать функции и внутри будет полученное в ней значение.

**Типы данных. Boolean.**

Очень важный тип данных, хоть и простой. **True, false**. Все.



В allowAccess в данном случае запишется false, потому что **age < 18**

В основном Boolean используется в условиях.

**Типы данных. null и undefined.**

Тип данных **null** формируется из единственного значения **null**. Он используется, чтобы пометить что-то как пустышку. Часто используют при инициализации переменных, которая изначально не имеет никаких значений.



Если же не указать в качестве значения null, то переменная будет хранить **undefined,**

**не смотреть на undefined после объявления**

Как видно по коду выше, когда я объявил переменную со значением **null,** ее тип данных стал **object,** а хранила она **null.** Хм.. больше нет такого типа данных как **Null?**



А когда я объявил переменную без значения, то ее тип стал **undefined** и хранила она **undefined.**

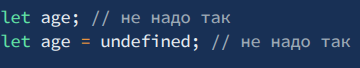
**Тип undefined**

В отличии от **null** он действительно является и типом, и значением. Дословно переводится как **не определено.** С точки зрения программирования можно интерпретировать как: **значение не присвоено.** То есть переменная объявлена, но не проинициализирована.

Может показаться, что **null** и **undefined** являются одним и тем же. Но это рассуждение ошибочно.

**Использовать undefined уместно только в тех случаях коде, где нужно проверить существует ли значение как таковое или была ли чем-то инициализирована переменная.**

Если хотим инициализировать переменную с пустым значением, то так **не делаем:**



Делаем так:

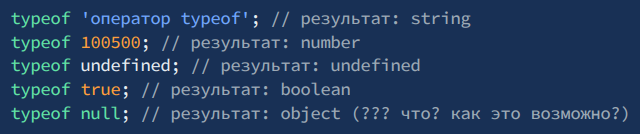


**Типы данных. Оператор typeof**

**typeof** это же функция, а не оператор. Нет? Ааа… погодите и да и да. Есть функция, а есть оператор.

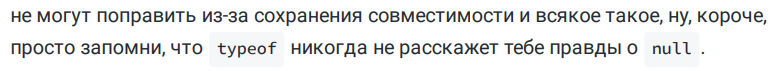


Этот оператор возвращает тип переменной.



О… здесь что-то об object?

Оператор и функция typeof почти всегда правдивы. Кроме ситуации с Null. В ней, они вместо типа данных Null показывают object. Это известная ошибка.



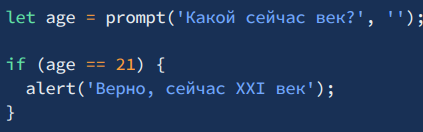
Рекомендуется вызывать этот оператор, как функцию. Но на деле разницы нет. Полезно проверять переменную через этот оператор, перед началом действия.

**Условные операторы: if и ?**

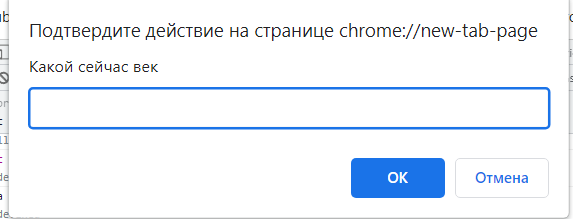
Условные операторы, нужны, чтобы обрабатывать условия…

есть два оператора: **if** и **?**

Вот интересный пример:



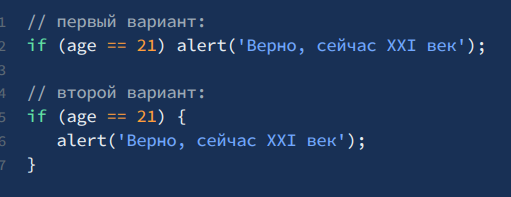
Функция **prompt()** тоже любопытная. Она позволяет записать в переменную значение, выведенное с клавиатуры, input() в питоне или cin в c++. Введенное значение будет иметь тип **String.**



Затем проверяется условие и если оно выполняется, то выводится верно… и тд.

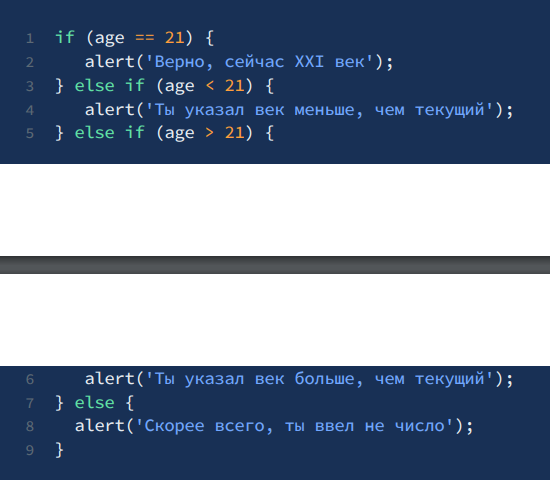
Есть два варианта, как написать действие, после условия. **В той же строке** или через **фигурные скобки.**

Без фигурных скобок после условия можно прописать только одну команду. Во всех остальных случаях надо писать фигурные скобки.



Но лучше использовать фигурные скобки **ВСЕГДА.**

**else** работает также как **if,** но срабатывает если **if** не сработал. После **else** можно также написать другой **if.**

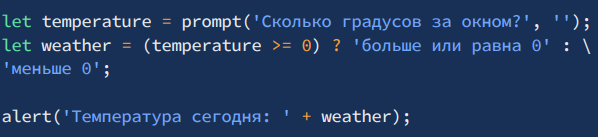


**Оператор “?” – Тернарный оператор.**

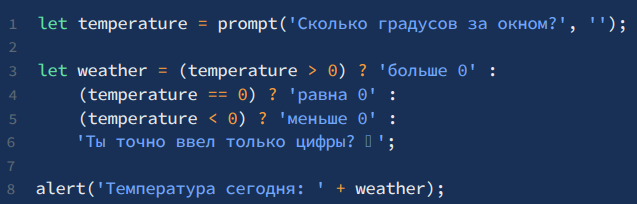
Тернарный оператор (состоящий из трех частей) – имеет **3 операнда.**

Как я понял: **условие, да, иначе.**

Пишется это таким образом: **(Условие в скобках) ? ‘Результат1’ : ‘Результат2’**



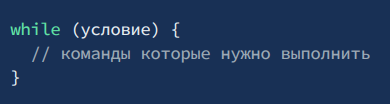
Запись полезная, но не привычная. И в иначе можно вставить еще одно если, Зато можно написать такой красивый код:

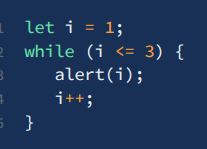


Надо переходить на такую запись.

**Циклы: while, for.**

Циклы нужны чтобы выполнять один и тот же код определенное количество раз. Синтаксис цикла **while:**



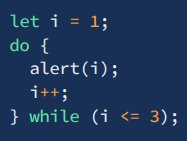


Выведет значения от 1 до 3.

Оператор **++** называется **инкрементом i++** иначе можно записать так: **i +=1**

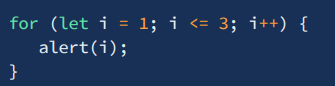
**Цикл do... while**

Логика у него такая же, но он сначала выполнит команды внутри цикла хотя бы один раз, а затем проверяет условие. Синтаксис у него такой:



Может быть полезным в редких случаях.

**Цикл for** один из моих любимых циклов.



Тупо удобный цикл.

**Операторы**

Большая их часть известна всем, кто изучал математику.

 **плюс, минус, делить, умножить, присвоить.**

Да присвоить, оператор равенства другой, сам знаешь.

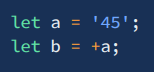
**Унарный/бинарный плюс.**

С помощью оператора **+** можно складывать не только числа, но и строки. Кажется это называется **конкатенация строк.**



Кстати, значения, к которым ты хочешь применить любой оператор, называются **аргументами** или **операндами.**

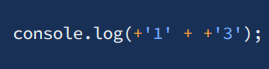
Чтобы избежать проблем со складыванием значений, нужно все приводить к одному типу.



Присвоение к числу с помощью **оператора +**

Это называется **унарный плюс**, **плюс применяется только к одному операнду.**

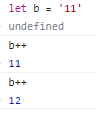
Вот другой пример такой записи:



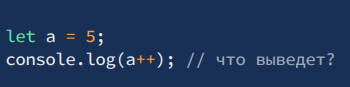
Ответ 4. Обе строки преобразовались в цифры и ответ будет 4. В рамках одной строки унарные плюсы имеют больший приоритет чем бинарные. Поэтому сначала произойдет преобразование типов, а уже потом они складываются.

**Бинарный плюс,** как уже стало понятно плюс, который применяется к двум операндам.

**Инкремент** – увеличивает текущее значение числа на единицу. Если применить его к строке. То он не увеличит значение строки, а просто преобразует тип, а уже при следующем применении увеличит значение.



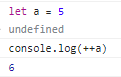
Такая запись называется постфиксная. Она применяется после. Пример:



Эта строка выведет 5. А уже после **a** будет равен 6.

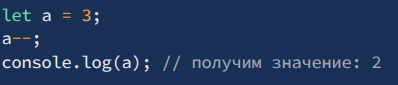
Чтобы сразу видеть 6, нужно использовать префиксную форму записи инкремента. **++a**

Пример:



**Декремент.**

Декремент работает точно также, только вместо того, чтобы прибавить, он убавляет.

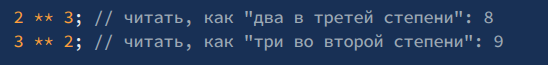


Одно замечание, инкремент и декремент можно применить только с переменными.

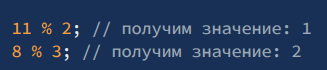
Если попытаться сделать это с обычными значениями словим ошибку.



**Возведение в степень.**



**Остаток от деления**



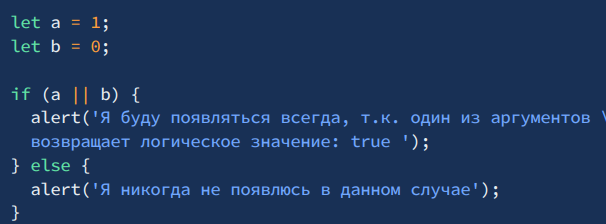
**Логические операторы**

Существует три логических оператора: И, ИЛИ, НЕ.

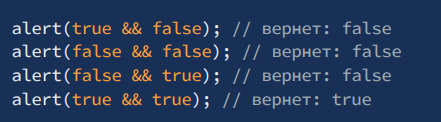
В коде: **&&, ||, !**

**||** или, всегда используется с 2-мя операндами. Это бинарный оператор.

Он выполняется возвращает истину если положительно хотя бы одно условие. Если же оба условия ложь, то и результат будет ложным.



**&&** и, такой же бинарный оператор. Но истину он возвращает, если истинны оба условие, во всех остальных случаях он возвращает ложь.

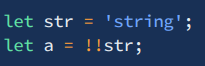


**Оператор ! (НЕ)** это унарный оператор. Он способен инвертировать условия.



И похоже он сразу приводит результат к типу **Boolean.** Таким образом как можно было использовать знак +, чтобы привести строку к типу числа, можно использовать оператор **!!** два раза, чтобы привести значение к типу **Boolean**, вместо функции **Boolean().**

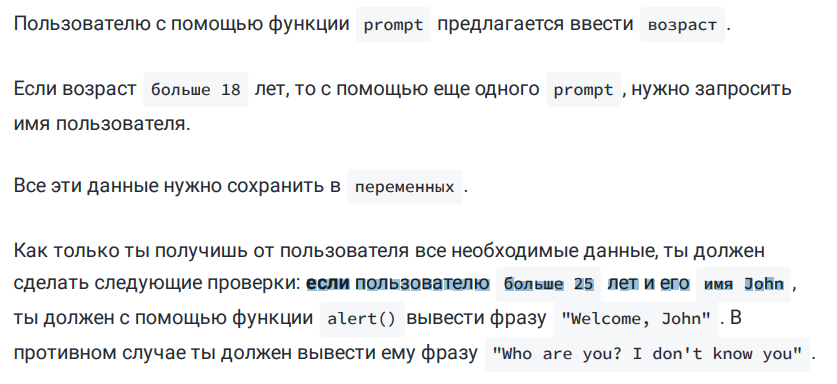
Лучше так

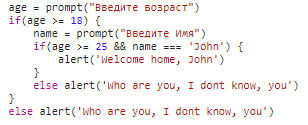


Чем так



**ДЗ**



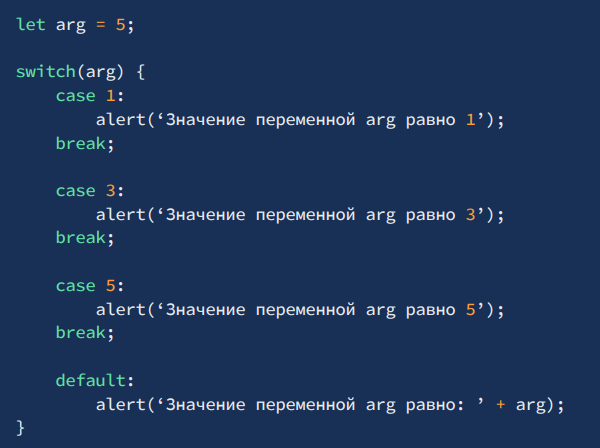


**Знакомство со switch**

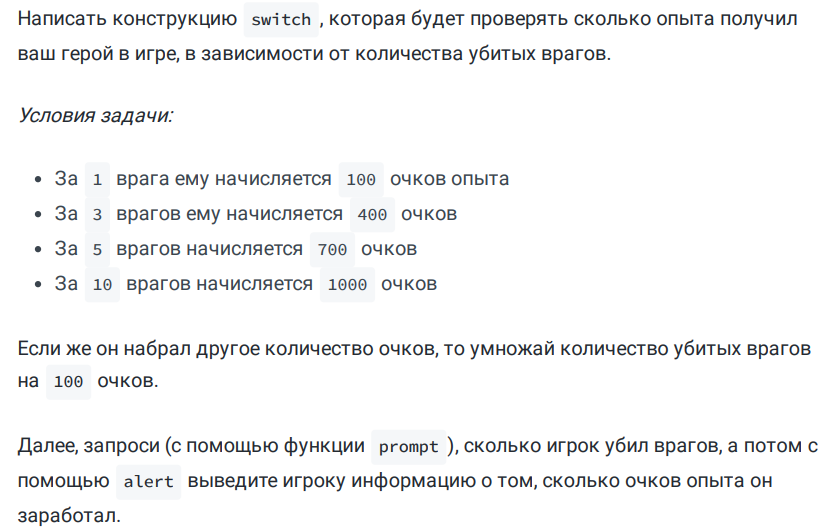
**switch** это более продвинутый оператор **if.** В спорах что лучше, **switch** или **if** лучше будет свитч. Подробнее. Свитч принимает аргумент. А case выступают в качестве условий, с определенным значением аргумента. Если же ни одно условие не было выполнено, то отработает условие **default, break** нужен чтобы выйти из **switch.** Блок **default** является **необязательным.**

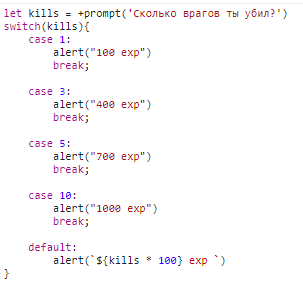
в case можно складывать не только числа, но и строки. Но и аргумент в таком случае тоже должен быть строкой. Иначе условия не отработают. В конце **default: break** не нужен, т.к. он и так всегда должен быть последним условием конструкции **switch.**

Пример:



**ДЗ**

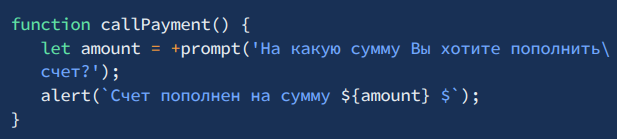




**Функции**

Зачем придуманы итак понятно, чтобы не плодить по проекту кучу повторяющегося кода.

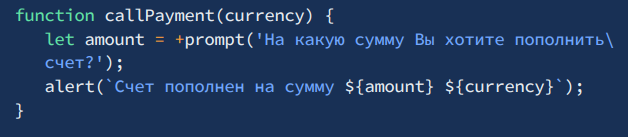
Пример синтаксиса. Объявление функции:

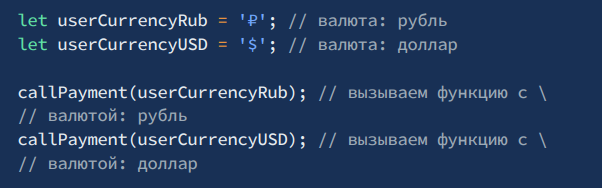


Вызов функции:



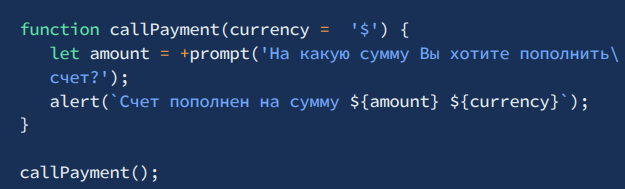
Пример функции с аргументом.





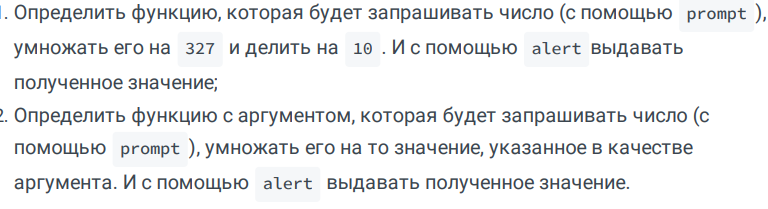
Но при использовании аргументов нужно учитывать такие моменты, когда пользователь вызывает функцию, но без аргументов. В таких случаях чтобы в функции не было значения undefined у передаваемому значению можно задать значение по умолчанию

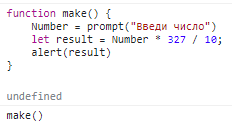
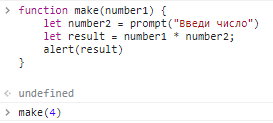
Пример.



Здесь currency по умолчанию доллар. Если же в функцию передать другое значение, то уже будет не доллар.

**ДЗ**

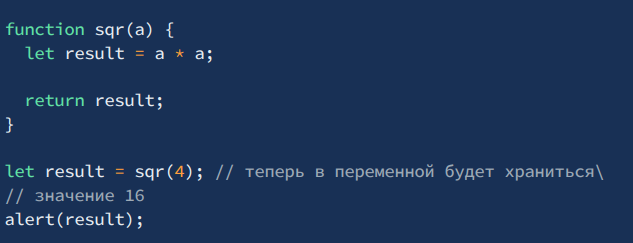


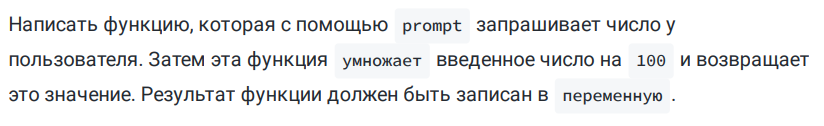
**Функции. Возврат значения.**

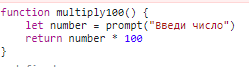
**Возврат нужен..**  чтобы функция что-то возвращала.

Пример



**ДЗ**



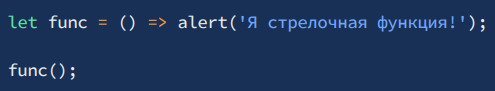




**Стрелочные функции**

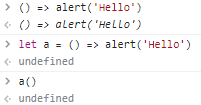
Данный тип функции пришел из **стандарта ES6.** Занимает меньше строчек, этим очень удобен.

Синтаксис.

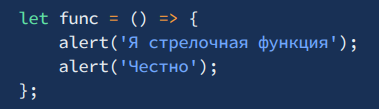


То есть смотря на эту запись можно понять, что функция присвоена переменной func, которая становится после этого функцией. А если не присвоить, то что произойдет? Я полагаю, должен произойти вызов.

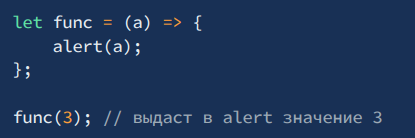
А вот и нет. Ничего не произошло. Вызова не было.



Это пример функции, когда нужно выполнить только одно действие. А если их больше, то нужно использовать фигурные скобки, и после каждого действия ставить точка-запятую **';'.**



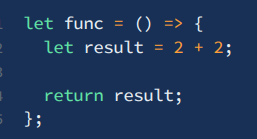
Вот пример передачи аргументов.

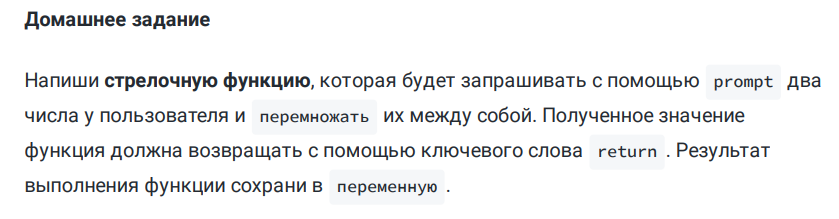


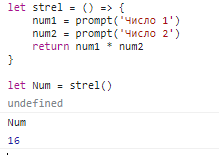
При возврате значения. Есть нюанс. Если функция делает только одно действие. То результат этого действия она и возвращает.



Если же действий больше, то уже не обойтись без фигурных скобок и слова **return.**

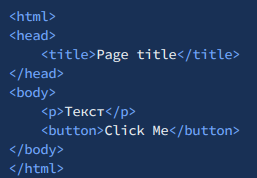






**Знакомство с DOM**

Все теги документа браузером рассматриваются как объекты. Внутри одних тегов могут быть другие, такие теги называются **дочерними**. И весь html по факту является деревом тегов с разными разветвлениями. Это и есть **DOM.**



Все, что существует внутри **html** документа, является частью DOM-дерева.

DOM позволяет менять объекты html документа местами и тд.

**DOM-элементы. Получение объектов.**

Чтобы изменить какой-то элемент. К нему сначала нужно как-то обратится. Это сделать можно по-разному. Но пока с помощью ID.

 Тег параграфа с id текст.

В одном документе не может быть элементов с одинаковым **id.**

Чтобы получить элемент по имени его **id** нужно использовать следующую конструкцию:

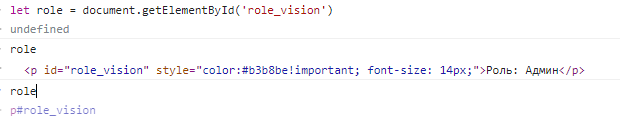


По порядку:

**document. –** говорит, что мы ищем элемент в документе.

**getElementById –** получить элемент с id…

В скобках указана строка с именем id.



Элемент получен. Теперь, чтобы изменить, например, цвет этого элемента можно обращаться к этой переменной.

В моем случае это:

**role.style.color = ‘green’**

Другой пример



Можно делать присвоение и без создания переменной:



Но использую переменную при повторном желании изменить этот элемент, будет проще. Не надо будет заново обращаться к элементу по ID.

**Вот таким образом можно применить сразу несколько стилей:**

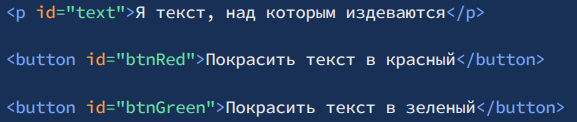


**DOM-элементы. События. Ввод.**

Каждое действие в браузере обрабатывается через Java Script. Движения мыши, нажатия на кнопки и тд. Сейчас речь пойдет о событиях.

Вот страница:





Текст и две кнопки. **ID btn** это видимо сокращение от слова **button.**

Так вот, при нажатии на эти кнопки текст должен менять свой цвет.

Для начало нужно получить в JS 3 этих элемента

**let text = document.getElementById(‘text’);**

**let btnRed = document.getElementById(‘btnRed’) ;**

**let btnGreen = document.getElementById(‘btnGreen’) ;**

Если ошибиться при вводе переменных, то в них запишется **null.**

**Детект события click.**

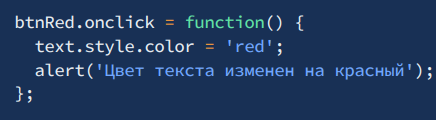
Два самых популярных способа обработки клика это:

Метод **onclick** у элемента button.

Метод **addEventListener**

**Метод onclick**

Пример

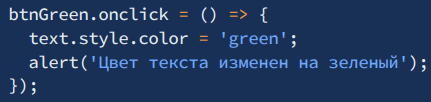


Имя переменной ‘**.onclick’**

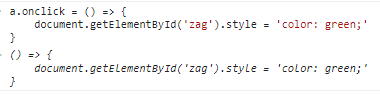
и присваиваем этому событию функцию.

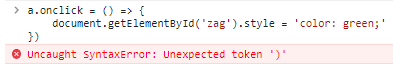
**Событие сработает только при клике левой кнопкой мыши.**

Лучше писать стрелочные функции.

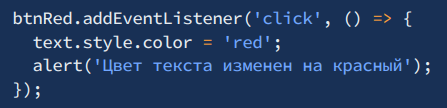


Черт, в таком методе написания… ты дурак автор книги? Ты зачем круглую скобку поставил в конце?

Нужно так!!!

Иначе будет так.

**Второй способ обработки:**



**А вот тут круглая скобка к месту. Так как открывается она после addEventListener(**

Уже отсюда видно, что эта функция работает понятно. В качестве первого аргумента выступает событие, которые мы обрабатываем, а в качестве второго аргумента выступает функция.

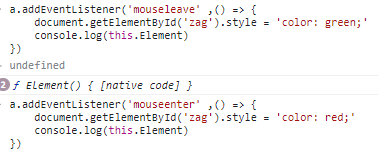
Пару примеров событий с мышью:

**mouseenter** – мышь нацелена на объект

**mouseleave** – мышь покинула объект.

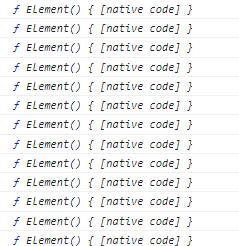
**mousemove** – мышь перемещается по объекту.

К примеру я прописал два обработчика на один объект:



И теперь, когда навожусь на объект, текст становится красным, а как убираю мышь, зеленым.

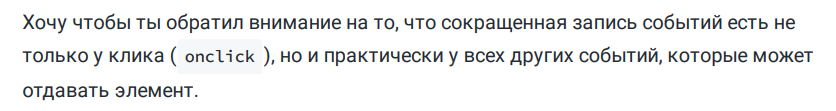
Правда консоль на каждое событие выводит сообщение.



С методом **onclick** такого не происходит. А.. вру. Такие сообщения появлялись только при событиях **mouseleave** и **mouseenter,** при событии **click** такого не было.

По факту разница между ними в том, что **addEventListener** более универсальный метод, который может принимать множество событий. Список есть тут:

<https://developer.mozilla.org/ru/docs/Web/Events>

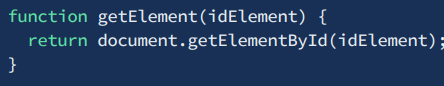


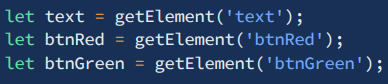
**Не совсем понял…** Теперь понял. Также как и для события **click** есть быстрая запись **onclick,** также и для других событий например для **mousemove,** есть метод **onmousemove.**



**Улучшение кода:**

Во первых постоянная **document.getElementById()** бесит. Проще переписать это в более простую функцию:

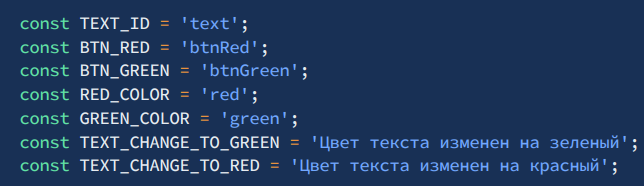


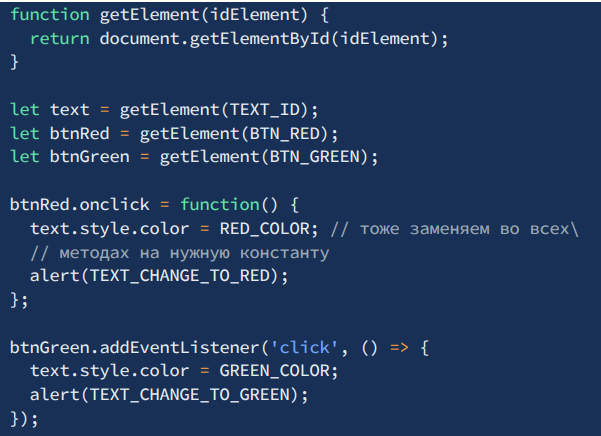


Как камень с души.

Во-вторых, у программистов есть **негласное правило –** все строковые значения, которые никак не будут меняться походу написания программного кода нужно выносить в отдельные переменные, так называемые **константы**.

И все константы прописываются не стилем **camelCase (**когда слова последующие слова пишутся с заглавной буквы без разделителей**),** а стилем **SNAKE\_CASE** с заглавными буквами:



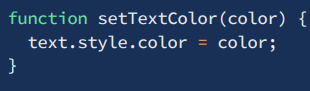


Вот это хороший, красивый код. **В реальных проектах константы определяются либо в отдельных файлах, либо в самом верху листинга кода.** Так быстрее получится отыскать нужную константу.

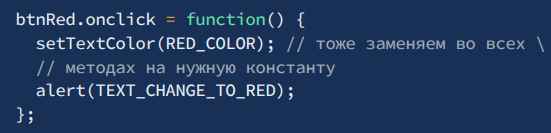
**Третье улучшение.**



Эту строку предложено заменить на функцию



И писать так:



Но это не очень разумно. А если я хочу изменять цвет другого тега? Не текст, мне писать новую функцию? Я бы доработал эту функцию несколько иначе:

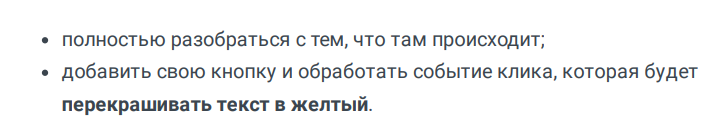
function setTextColor(tag, color){

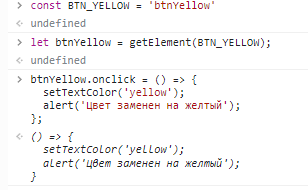
tag.style.color = color;  
}

Тогда функция была бы более универсальной

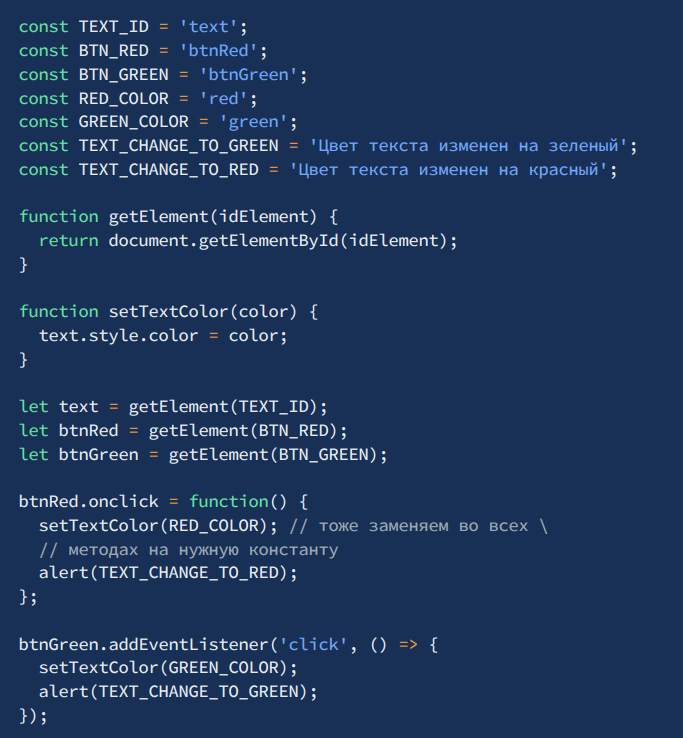


**ДЗ**



**Вот, даже сохранять не буду. Тупо в консоли.**

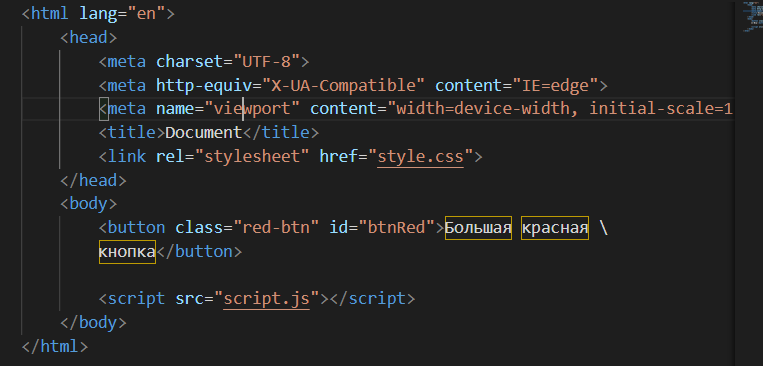
**А так вот что у книги получилось. Норм.**



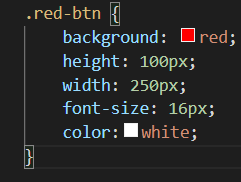
**DOM – элементы. События. часть 2.**

Более подробно о всех событиях. Хватит писать в консоли. Создам 3 файла с таким содержимым:

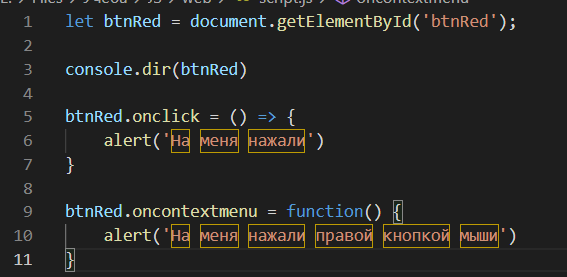
**index.html**



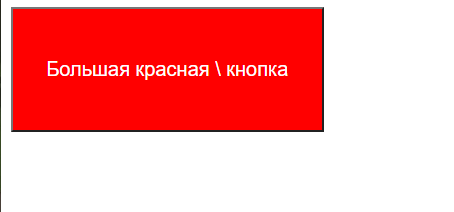
**style.css**



**script.js**



**Открытая страница.**



Что за \ после слова красная? А я не знаю. Может потому что в коде, который я переписывал был таким?



Разберем js.

Сначала записываем в переменную элемент с **id=’btnRed’**

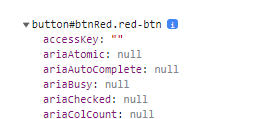


Далее строка:

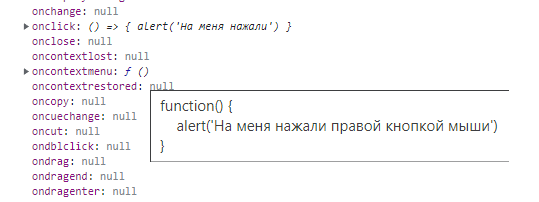


**console.log ()** выводит в консоль текстовое сообщение. А **console.dir ()** выводит список свойств указанного **js-объекта.**

**Вот часть:**



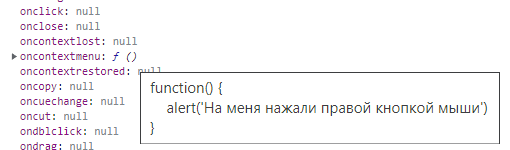
Теперь в этом списке нужно найти те свойства, которые начинаются со слова **on.**

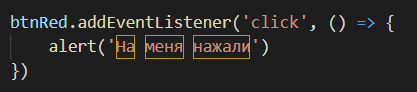


И как можно заметить, на этой кнопке по событию **onclick** записана стрелочная функция, а по событию **oncontextmenu** записана обычная функция, которую полностью видно только если на нее навестись.

В это и суть. Все события, которые здесь есть можно обработать, как метод **onclick,** так и через метод **addEventListener(),** но уже без приставки **on**.

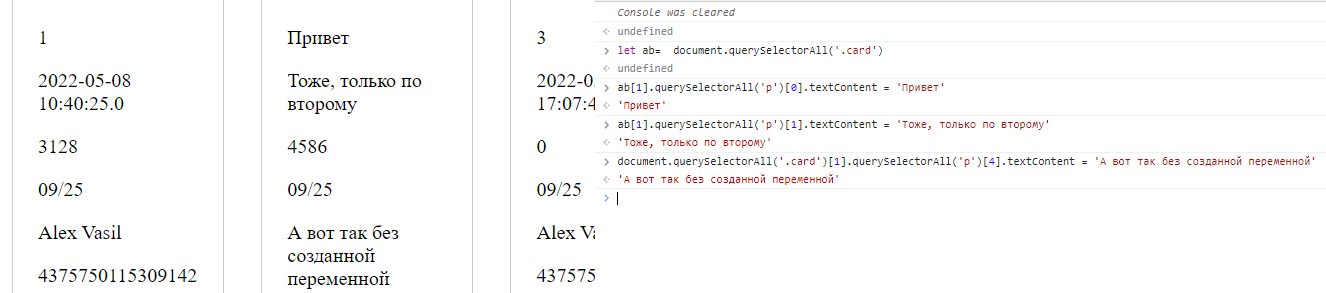
Если использовать метод **addEventListener(),** то событие **‘on’** будет пустым.

**onclick пустой**

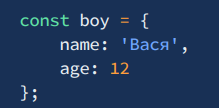


**Объекты**

Но перед этим хочу сохранить картинку, как я научился обращаться к элементам

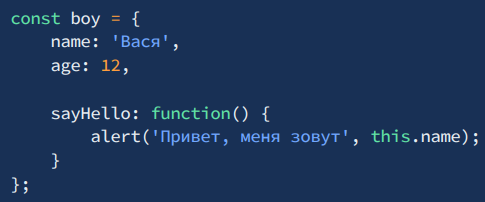


Для начала, вот пример объекта.

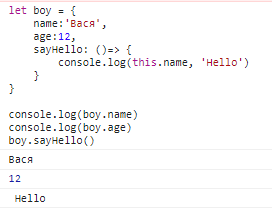


поля **name** и **age** это **свойства объекта.** В свойствах можно записывать любые типы данных, включая массивы и сами объекты.

А вот про то, что в них можно засунуть методы это я не знал. Вот:

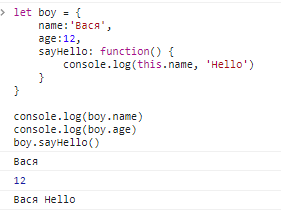


Попробовал написать тоже, но со стрелочной функцией.



Но имя функция не пишет, только hello

С обычной функцией работает.

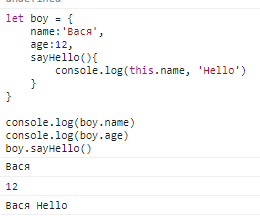


Дело в том, что

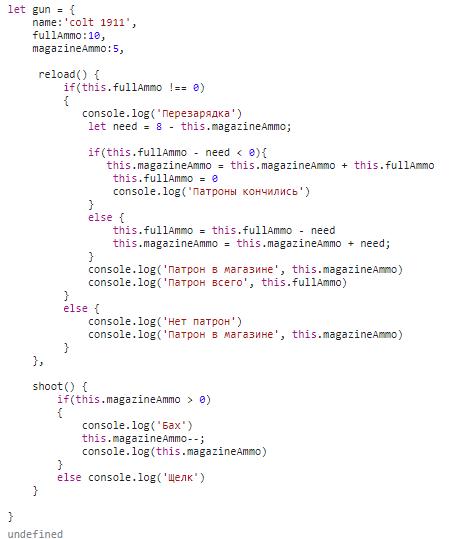


Поэтому если есть обращение, то лучше использовать другую функцию.

Но есть более короткий способ написания обычной функции в объекте:

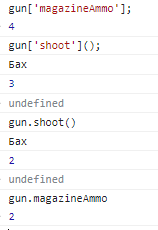


Хочу создать стреляющий объект пистолет на 8 патрон.

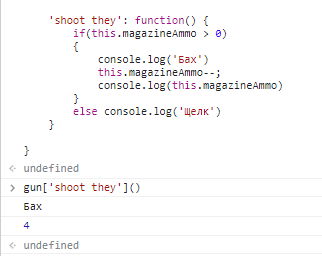


Ностальгия. Помню похожее писал на C++.

Есть два способа обращаться к полям и методам объекта. Вот они:

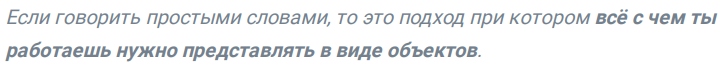


Мне больше нравится через точку. Второе обращение имитирует обращение к элементам массива. **При вызове через точку есть только одно правило. Название свойства не должно содержать пробелов.** Если название свойства к примеру, является строкой. То такое свойство можно будет вызвать только через квадратные скобки.



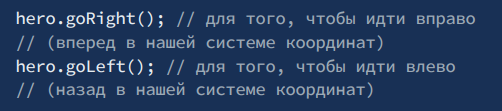
Но методы все же принято называть через **camelCase.** Так что это скорее может возникнуть при генерировании объектов с определенными именами.

**JavaScript** – **объектно-ориентированный язык.** Объектно-ориентированного подхода придерживаются почти все программисты.



А вот такой код написан в книге.





Он дописал еще пару методов, но они такие же.

Подведем итог.

В **Свойства** записываются какие-либо данные относящиеся к **объекту.**

В **Методы** предназначены для того, чтобы что-то сделать со **свойствами.**

**This** необходим чтобы внутри **методов** обращаться к **свойствам** или другим **методам** внутри самого объекта.

**Массивы**

Массивы очень важный тип данных. Синтаксис.



Вот два таких способа записи. **Но лучше использовать второй. Это лучшая практика среди разработчиков и так быстрее.**

Массивы могут быть наполнены любыми типами данных.



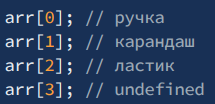
И совмещать в одном массиве разные типы тоже можно.

У любого массива есть **свойство length**(длина)

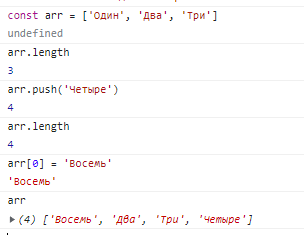




Массивы итерируемые. А значит можно обратится к любому элементу массива по индексу.



Также не важно константа массив или обычная переменная, его все равно можно изменять.



Выше описан метод **push**, который добавляет в конец массив новый элемент. Но есть и другие способы сделать это. Например, можно обратится к элементу массива после последнего и записать туда что-то.

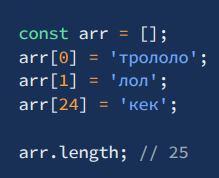


**При этом если обратится к какому-нибудь 10-му элементу и записать туда значение. То это значение запишется в 10 элемент, а длина массива увеличится до 11. При этом в массиве появится несколько переменных со значение undefined.**



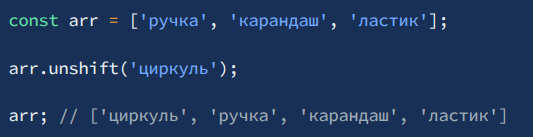
**Но таким образом не рекомендуется добавлять переменные в массив.**

Здесь как раз об этом. И почему так делать не стоит.



**Метод push** уже понятен, но что если объект нужно добавить не в конец, а в начало массива?

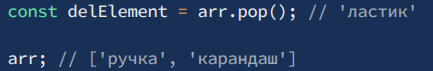
В таких случаях очень полезен метод **unshift.**



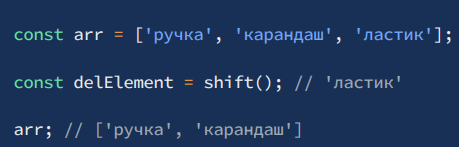
Для удаления из массива последнего элемента используется метод **pop.**

У этого метода есть одна фишка. Этот метод не только удаляет последний элемент, во время удаления **метод** возвращает удаленное значение. Так можно определенные значения вытаскивать и сохранять.





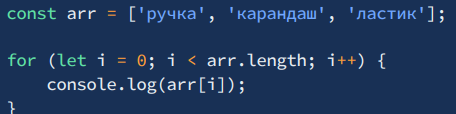
Точно также работает метод **shift.** Только удаляет он значения не в конце, а в начале.



**Перебор элементов массива.**

Чтобы перебрать элементы массива есть 2 способа. **Циклы** и встроенный метод **forEach.**

**Цикл for** является старым способом. Но бывает иногда крайне необходим.

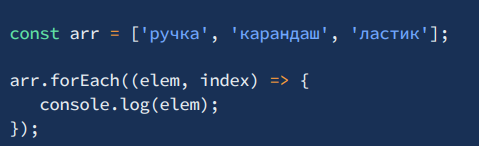


Второй способ **Цикл for … of.**

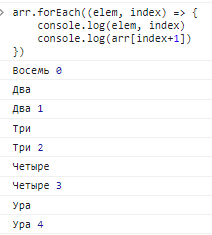


В **python** был такой цикл. У этого способ минус в том, что в этом цикле нельзя получить индекс элемента в массиве. Но он нужен не всегда. Поэтому тоже имеет право на жизнь.

Третий способ **Цикл forEach**

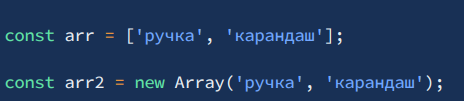


С этим способом у меня были какие-то проблемы. Нука… Ну в целом, можно обращаться к переменным напрямую через elem или как его назвать, так и с помощью названия массива и индекса.

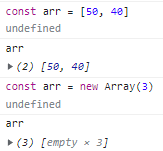
Этот метод не очень люблю из-за того, что в него нужно класть функцию.

**Почему не стоит использовать new Array()**

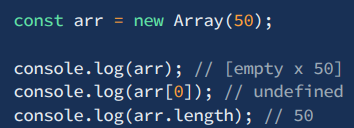
Причина 1 – Длинная запись.



Причина 2 страшнее. Похоже функция **Array** не очень дружит с одиночными цифирными значениями.



**Да, new Array(3)** создал массив с 3-мя **undefined** значениями.

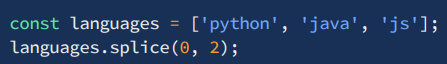


**Методы массивов: splice, slice, concat.**

Разберем оставшиеся методы массивов.

**Метод splice.**  Удобный метод для **добавления** и **удаления элементов.**

**Удаление:**



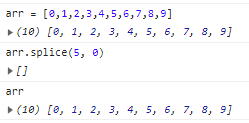
Первый аргумент, индекс с которого начать удаление, второй аргумент, количество удаляемых элементов. В данном случае из этого массива останется только **js.**



Если ввести только один аргумент, то удаляться все элементы начиная с этого индекса.



А если в качестве второго аргумента указать 0, то из массива не будет удален никакой элемент. **Зачем это нужно чуть ниже.**



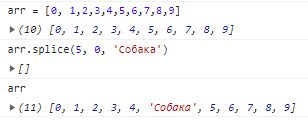
**Добавление элементов.**

Чтобы добавить элемент в массив, нужно после первых 2 параметров указать добавляемые значения. Их может быть любое количество не зависимо от того, сколько переменных было удалено.



И как видно, они вставляются именно в то, место массив, который указан в качестве первого аргумента.

**А если в качестве второго аргумента указан ноль, то можно без удаления вставить в массив свои значения без удаления и со сдвигом.**



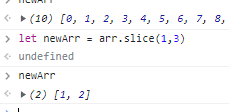
Хочется добавить, что как также как методы **pop** и **shift,** метод **splice** возвращает удаляемые значения. И все эти методы изменяют исходный массив. По этому нет смысла писать:

**arr = arr.splice(1, 2)**

**Метод slice**

Метод копирует указанный участок массива и создает новый.

Первый аргумент с какого начать, второй аргумент до какого копировать (**Не включительно**).

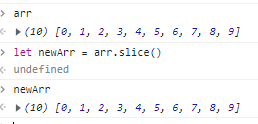


В примере выше копирование начинается с первого индекса числа **1,** до 3-го индекса, числа **3**, **но не включая это число.**

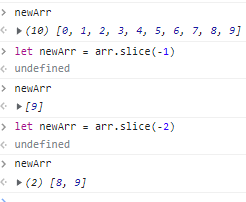
Если указывать только один элемент. То охватится весь массив, начиная с этого элемента.



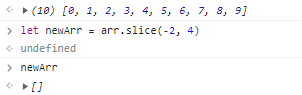
А если вообще не указывать аргументы, то массив скопируется целиком.



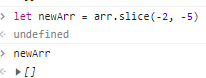
**Аргументы могут быть также отрицательные.**



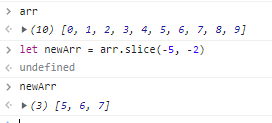
Но если указывать отрицательные аргументы, они оба должны быть отрицательными. Иначе, ничего не запишется.



И идти в порядке возрастания индекса тоже ничего не запишет.

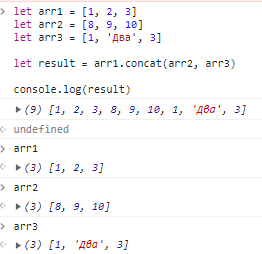


**Вот как правильно это делать.**

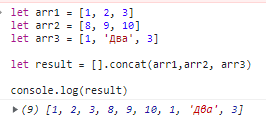


**Метод concat**

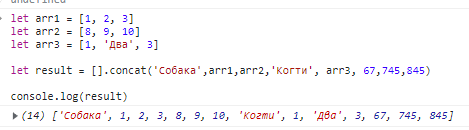
Складывает массивы в один. Не изменяет оригинальные массивы.



**Можно написать так:**



Если аргумент не является массивом, то он просто добавляется.



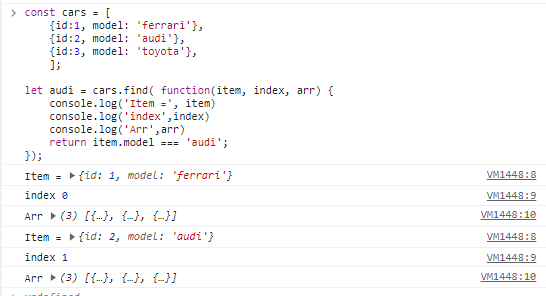
**Методы массивов: find, findIndex, filter**

**Метод** **find** позволяет найти значение первого элемента, найденного в массиве, который удовлетворяет условию, переданному в **callback-функцию.**

**callback-функция** запускается для каждого элемента в массиве. Если элемент с указанным условием не найден, то функция возвращает **undefined.**

Сложность этого метода в том, что он должен принимать функцию в качестве аргумента. **Чаще всего** **используется с массивами**, элементы которого являются объекты.

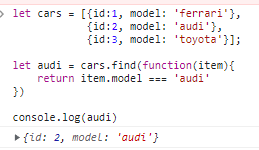
**Пример:**



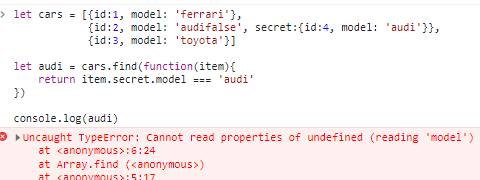
Как можно заметить, в **callback-**функцию можно засунуть 3 аргумента (2 последний используются редко). Первый будет содержать, рассматриваемый элемент, второй индекс этого элемента, а третий весь рассматриваемый массив.

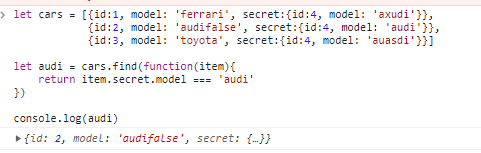
Функция **callback** будет применятся ко всем элементам массива, до тех пор, пока не будет найден тот, что прописан в return.

В принципе можно вызывать функцию и без этих дополнительных аргументов. Но зато теперь понятно, за что эти аргументы отвечают.

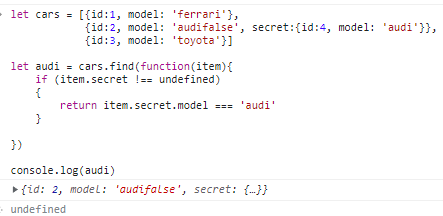


Если искомым объект находится в объекте внутри объекта. Например, внутри объекта secret. То метод find можно будет применить, только если у всех объектов будет такое же поле, иначе метод вызовет ошибку.

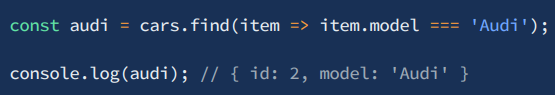


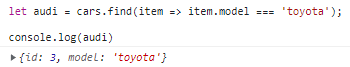


Однако можно добавить простую проверку, и тогда эту ошибку можно будет обойти.



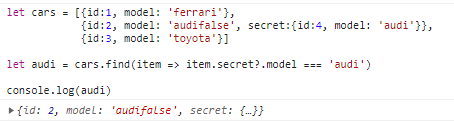
Чаще всего этот метод записывают со стрелочной функцией вот пример:





Но как сделать такую же запись с сохранением условия???

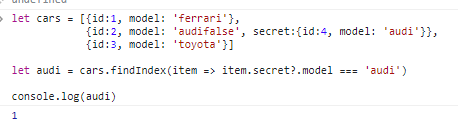
Знакомый подсказал. Это называется **optional chaining.** Достаточно поставить вопросительный знак перед опциональным объектом.



**optional chaining** можно использовать несколько раз, если необходимо.

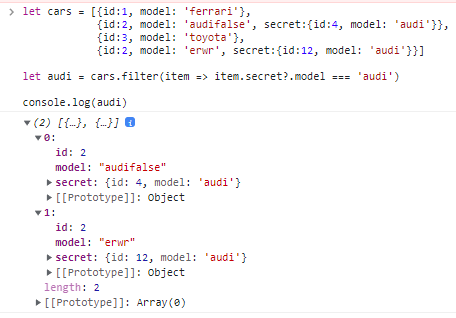


**Метод findIndex** работает точно также. Только вместо элемента, возвращает его индекс.



**Метод filter**

Данный делает тоже что и метод **find,** но он не останавливается после первого найденного элемента тем самым возвращая все элементы, удовлетворяющие условию.

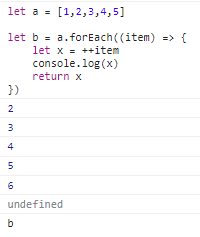
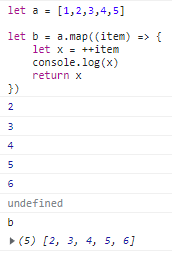


Или вот пример вывода все четных элементов.



**Методы массивов: map, sort.**

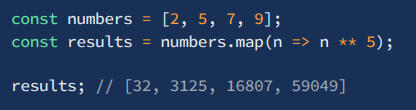
**Разница между методами forEach и map в двух картинках.**



Именно так. **Map** работает точно также, как **и forEach**. Но если делать в **map return**, то этот элемент будет записывать в новый массив. Таким образом **map** может создать измененную копию массива.

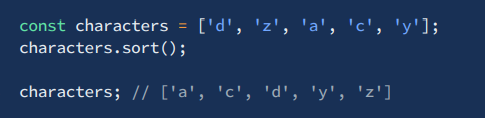
Внутри **map** может быть только один элемент, это **callback** функция, а она в свою очередь уже имеет 3 аргумента, которые уже разбирали.

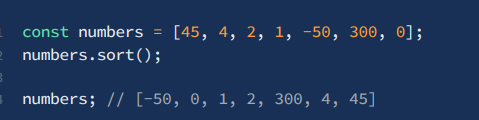
Но все же покажу пример из книги.



**Метод sort.**

Метод сортирует массив. **Причем изменяет исходный массив.** И возвращает измененный массив.





**Опа, а я и не заметил.** При сортировке массива с числами число 300 не на своем месте. Хм…

Вот причина:

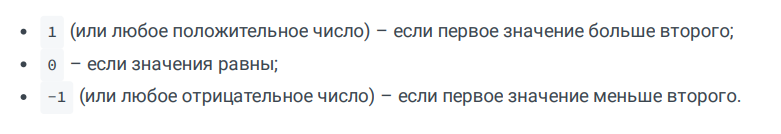
 

Этот метод приравнивает сравниваемые элементы к строкам. Как избежать этого?

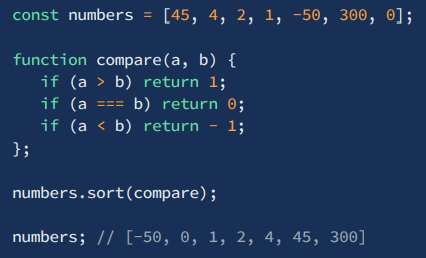
А в книге тут опечатка. Он должен был написать **меньше.**



Метод может принимать **callback-функцию,** которая должна задавать правила сортировки.



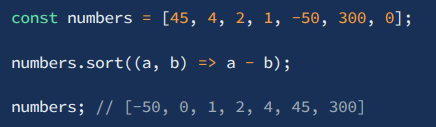
Вот объяснение.



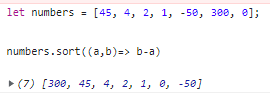
**Хочу поиграться чутка с этим.** Например, перепишу эту функцию чтобы сортировка была по убыванию.



Ааа как хитро. 1 0 -1 это не обязательные возвратные параметры. Вот как можно сократить эту функцию еще лучше.

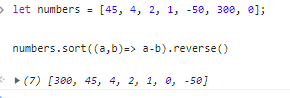


**И соответсвенно:**

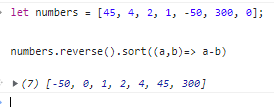


**Методы массивов: reverse, split, join.**

**Метод** **reverse** переворачивает массив. То есть можно не переписывать метод **sort,** а просто после него вызвать метод **reverse.** И получится массив по убыванию.



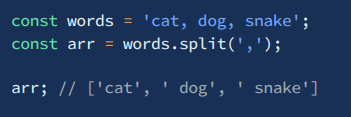
При таком написании, хочется отметить, что методы вызываются в том порядке, в котором написаны. Слева направо. Так что если написать так:



То массив сначала инвертируется, а уже потом от сортируется. Метод **reverse** изменяет текущий массив. (**Метод отрабатывает по месту**)

**Метод split** позволяет из строки получить массив. Метод не возвращает строку в форме массива. В качестве аргумента принимает **символ разделителя.**

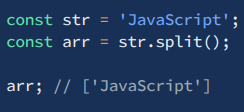
Пример:



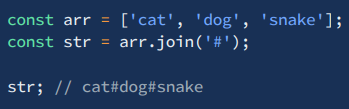
Если засунуть в качестве аргумента пустую строку, то метод разобьет изначальную строку посимвольно.

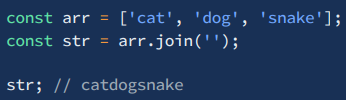


А если не передать ничего, то вернет всю строку, в виде единственного элемента массива. Таким образом этот метод можно использовать, чтобы преобразовать строку в массив с одним элементом.

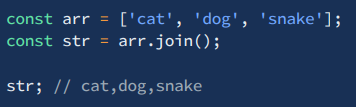


**Метод join** обратный метод **split.** Переводит массив в строку с указанным разделителем.





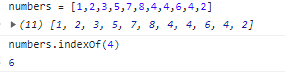
А если не указать разделителя, то будет строка, выглядящая как массив. То есть разделенная запятыми внутри.



**Методы массивов: indexOf, lastIndexOf, incudes**

Как же у массивов много методов…

**Метод indexOf** позволяет найти индекс элемента **по значению.**



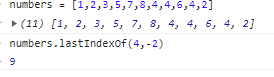
Как можно заметить, метод находит индекс **первого** найденного значения.

Если метод не находит искомого элемента, то возвращает **-1.**



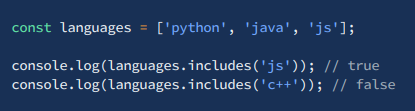
**Помимо первого аргумента** у метода есть второй необязательный аргумент. Индекс, с которого необходимо начать.

**Метод lastIndexOf** работает абсолютно также, но начинает поиск с конца массива. Только значения для второго аргумента нужно передавать отрицательное.



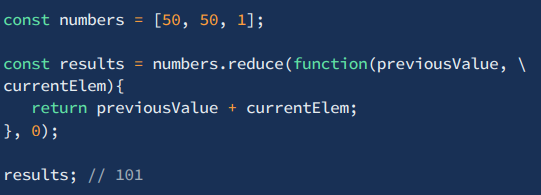
**Индекс возвращается не отрицательный если что.**

**Метод includes** проверяет, существует ли указанное значение в массиве. Первый аргумент, искомый элемент. Второй аргумент индекс с которого начать.

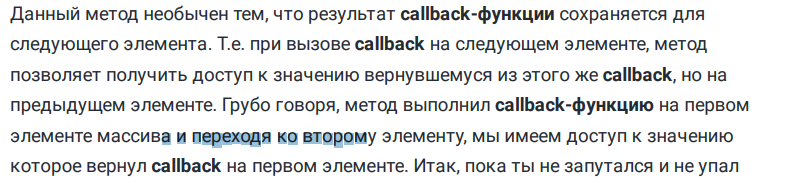


**Методы массивов: reduce, spread**

**Метод reduce, пожалуй,** один из самых сложных методов массивов. Вот пример.

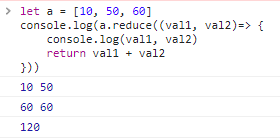


В книге не очень понятное объяснение.

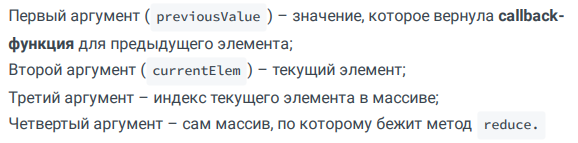


**Как бы я это объяснил.** Метод принимает **callback функцию.** У этой функции должно быть 2 аргумента. В первой итерации это **первый элемента** и **второй элемент** массива. В данном примере функция возвращает их сумму. А уже во всех следующих итерациях в качестве **первого аргумента** выступает значения **return** из прошлой итерации. А второй аргумент, это элемент следующий элемент массива, который в прошлой итерации не учувствовал.

Короче, мое объяснение похоже не лучше. Но как функция метод работает я понимаю. Сначала я даже подумал, что в самую первую итерацию **previousValue** равен нулю и затем после сложения с первым элементом, он просто не изменяется. Однако если бы это была бы операция умножения, то это логика бы разрушилась. Поэтому я решил проверить что выводят логии в каждой операции.

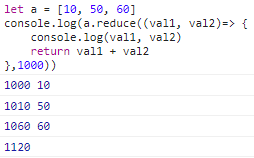
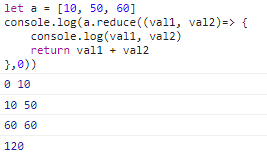


Видно, что в самом первом цикле значения действительно равны первым двум элементам. Во втором цикл уже складываются сумма из прошлых элементов и текущий. Последнее 120 это вывод. **Помимо двух этих параметров callback-функция может принять еще 2.**

****

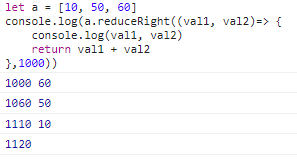
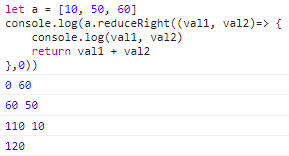
**И это еще не все. Сам метод reduce(func(), 2) помимо функции может принимать второй параметр (initialValue). Начальное значение в первой итерации.** Вот два примера со значениями 1000 и 0.

Важно помнить, что второй параметр указывается после закрытия **callback** функции, т.к. она первый аргумент.

Но это для простого сложения. А для других операций с тем же нулем могло быть все иначе.

У этого метода есть **инвертированная копия.** Работает аналогично, но справа налево. **reduceRight.**

** **

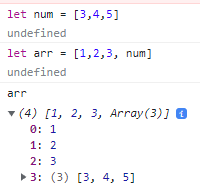
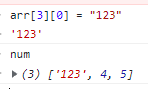
**Оператор spread.**

Это не метод массива. Но он полезен. Этот оператор позволяет объединять массивы. Он представляет из себя **…** 3 точки, которые ставятся перед переменной… Хм.. а если поставить после?



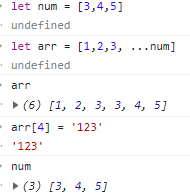
А понятно.

Вот что будет без него:

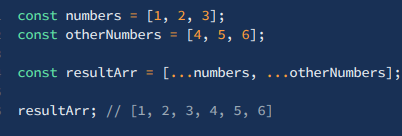
Вставленный массив изменяет и оригинальный, потому что вставленный массив является ссылкой на оригинальный массив.

А вот, что будет если использовать оператор **spread.**

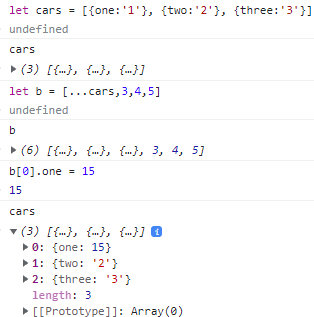
****

**Все равно, что метод concat.**

Метод не ограничивает в количестве использований. Можно легко получить копию массива. **Но…**



**Нельзя копировать массивы со сложными объектами. Почему?**

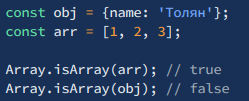
****

**Потому что изменяя этот сложный объект внутри нового массива изменятся будет и оригинальный.**

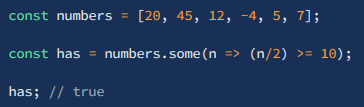
**Методы массивов: Array.isArray, some, every**

Так.. это предпоследняя глава про методы массивов. Как их много. Но я столько не знал.

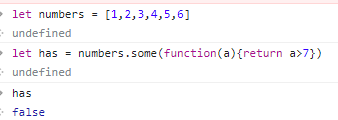
**Метод Array.isArray** служит проверкой, является ли проверяемый объект массивом. Если да, то возвращает **true,** иначе **false.**

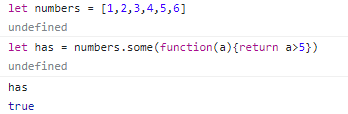


**Метод some** проверяет есть ли в массиве хотя бы один элемент, который удовлетворяет условию.



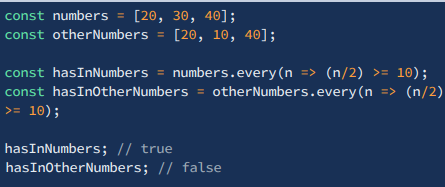
Вот пример без стрелочной функции.





В этот метод нужно вставить **callback** функцию с одним параметром.

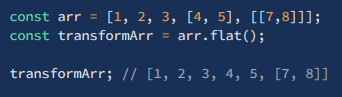
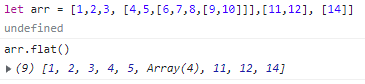
**Метод every** работаетаналогично. Но, как можно догадаться, **true** он выдаст только в том случае, если все элементы будут удовлетворять условию.



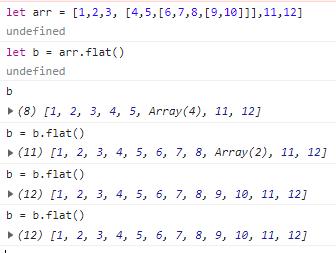
**Методы массивов: flat, fill.**

**Метод flat** относительно новый метод, но действительно удобный. Он служит для распаковки массивов внутри массива, возвращая новый массив.

Если использовать его без параметров, то он распакует ~~только 1 массив~~ массив на одном уровне вложенности.

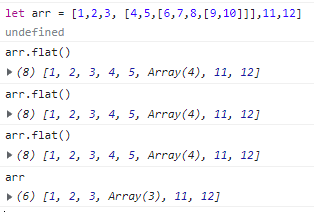
 

Вот как работает порядок распаковки.

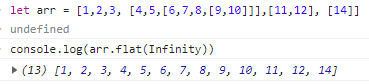


Если в массиве нет вложенных массивов, то метод возвращает исходный массив без изменений.

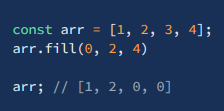
**Метод не изменяет исходный массив.**

****

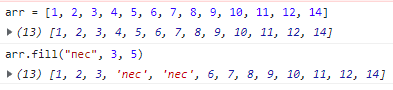
**Уровень сглаживания** вложенности по умолчании равен 1. Но его можно задать явно. Для этого в метод **flat** нужно передать число. Если неизвестен уровень вложенности в массиве, но важно распаковать их полностью. То можно воспользоваться числом **infinity.**

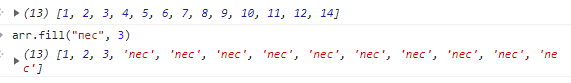


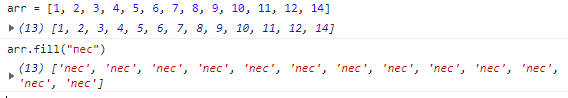
**Метод fill** заполняет массив значениями. Первый параметр это само значение.



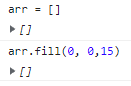
Второй параметр индекс с которого начать заполнение. Третий параметр индекс до которого заполнять, **не включая этот индекс.**







**Этот метод изменяет текущий массив.** Но этот метод не способен добавлять в массив новые элементы.

****

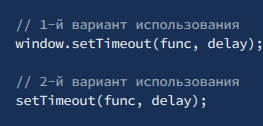
**Фух… методы массивов кончились. Надо собрать их в одну таблицу.**

|  |  |
| --- | --- |
| **push** | Добавить элемент в конце массива |
| **unshift** | Добавить элемент в начало массива |
| **pop** | Удалить последний элемент массива |
| **shift** | Удалить первый элемент массива |
| **forEach** | Обход массива без возврата значений |
| **new Array()** | Функция создания массива, если внутри одно число будет создан пустой массив такой длины. |
| **splice** | Метод для добавления, удаления и замены элементов. |
| **slice** | Метод копирования массива |
| **concat** | Метод объединения массивов |
| **find** | Найти значение первого элемента, найденного в массиве, который удовлетворяет условию |
| **findIndex** | Найти индекс первого элемента, найденного в массиве, который удовлетворяет условию |
| **filter** | Найти значение всех элементов, найденных в массиве, которые удовлетворяет условию |
| **map** | Аналогично forEach перебирает массив, но возвращает измененную копию массива |
| **sort** | Метод сортирует массив, метод сортировки заложен в callback функции |
| **reverse** | Перевернуть массив |
| **split** | Разбить строку на массив по указанному разделителю |
| **join** | Разбить массив на строку по со вставкой указанного разделителя |
| **indexOf** | Найти индекс элемент по значению, возвращает первый найденный |
| **lastIndexOf** | Найти индекс элемент по значению, возвращает последний найденный |
| **incudes** | Проверяет присутствие искомого элемент в массиве. |
| **reduce** | Позволяет работать над элементами массива, отправляя результат функции с следующую итерацию |
| **…spread** | Оператор, позволяющий вставлять в массив содержимое массива |
| **Array.isArray** | Функция проверки является ли переменная массивом. |
| **some** | Метод проверки присутствует ли в массиве хотя бы один элемент удовлетворяющий условию |
| **every** | Метод проверки, удовлетворяют ли все элементы массива условию |
| **flat** | Метод распаковки массивов в массиве |
| **fill** | Метод замены уже заданных значений массива на новые |

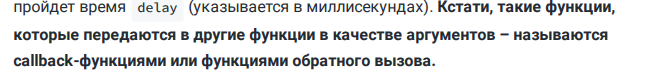
**Планирование вызова функции: setTimeout**

**setTimeout** это функция с двумя параметрами, другой функцией, которая будет выполняться. И аргументом паузы.

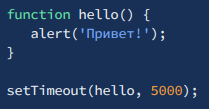
Есть два способа вызвать эту функцию.

****

**Рекомендуется использовать второй вариант.** И он короче.

****

Пример вызова. Важно обратить внимание, что передаваемая функция, внутри **SetTimeout,** вызываетсябез скобок().

****

**Если в SetTimeout передать функция так:**

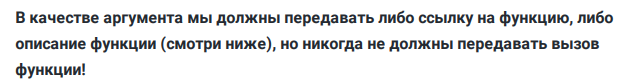


То во первых функция hello вызовется без задержки. А во вторых будет ошибка.

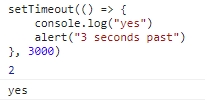


**В данному примере передалась ссылка на функцию:**

****

****

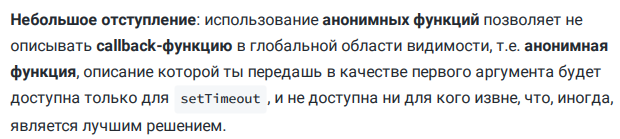
Описание функции… если я правильно понимаю, то это реализация функция прямо внутри **SetTimeout.** Да это именно оно.







Напомню, такие функции называются **анонимными.**



Не знаю в каком это случае это может быть лучшим решением. Но верю на слово.

А вот это очень интересно. В эту функцию можно передать строку и она выполнится.



Хотя в плане безопасности это плохой вариант, поэтому не рекомендуется.

Да и не получилось у меня.

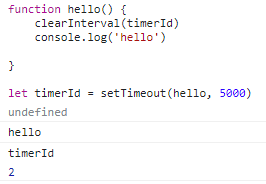


Может в другом браузере? Не.. Ну ладно. Жаль.

**Важно отметить,** метод **setTimeout** необходимо удалять во зимбежание **утечки памяти.** Чтобы сделать это нужно… Ща пгд. **setTimeout** помимо того, что выполняют **callback** функцию с заданной задержкой, еще и возвращает свой идентификатор.

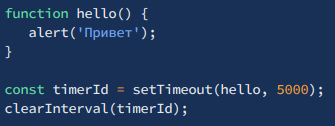


Но удалить его нужно только после того, как он отработает. Поэтому удаление стоит добавить внутри callback функции. Хоть в начале без разницы, так как она уже вызвалась.

Но блин… для каждого timeout создавать переменную.. бе…

Если же попробовать написать таким образом. То функция не выполнится вовсе.



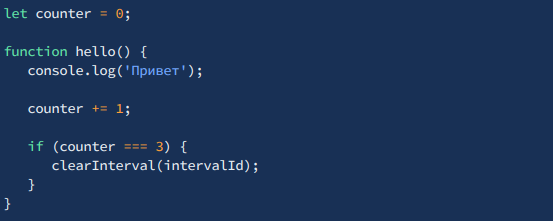
Так, как не успело пройти 5 секунд, как интервал был удален. И ничего не изменется даже если задержка будет равна 0.



**Планирование вызова функции: setInterval**

Работает аналогично **setTimeout,** только вместо одного раза будет вызываться до тех пор, пока **timerId** не будет удален.

Вот пример функции, которая будет вызываться 3 раза с интервалом в секунду.

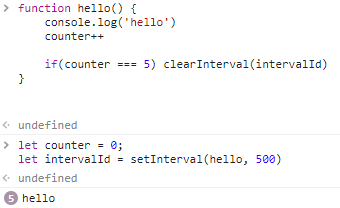




1



Аааа….



**Объекты. Свойства**

В прошлых уроках рассматривали примитивные типы данных (все кроме массивов). Важные типы данных. Однако, **JavaScript является объектно-ориентированным (ООП) языком,** и в нем часто используются **ООП подходы.** Не уметь применять эти подходы это упущение. Его надо исправить.

**Объект –** сложная сущность, которая может хранить в себе целый набор различных данных, примитивные значения, массивы и функции.

Создать пустой **объект** можно 2-мя способами.

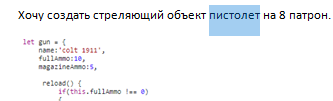


Первый вариант называется **конструктором объекта.**

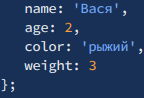
Второй вариант называет **литералом объекта (**литеральная нотация**)**

В принципе ничего плохого ни в одном из вариантов нет. Поэтому может спокойно использовать тот, что короче.

Напомню, что объект я уже создавал. И книга уже показывала на примере героя. Надеюсь это не просто повторение.









Пока что баян. Получение цвета кота.



В качестве имен свойств могут выступать и строки. Хотя я уже говорил.

**Рекомендуется делать название без кавычек.** Кавычки можно использовать если в качестве названия например будет выступать имя какой-то переменной, которая создаться в этом объекте. По крайней мере я так делал однажды. Но к такому свойству можно будет обратится только через **квадратные скобки с кавычками.**

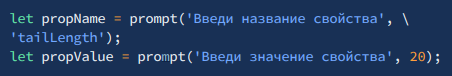




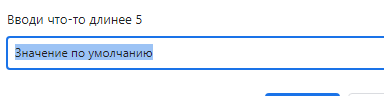
Ну… пробелы использовать не буду, но кавычки буду при необходимости.

**В объекте можно определить свойство, имя которого заранее неизвестно.** Хм.. любопытно.

Допустим, попробуем добавить в объект имя поля и значение введенные пользователем..



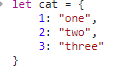
Оказывается функция prompt может принять второй аргумент. Это значение по умолчанию которое будет в поле … по умолчанию.

****

Но это если нажать отмена, в поле ничего не запишется. На это значение по умолчание не влияет.



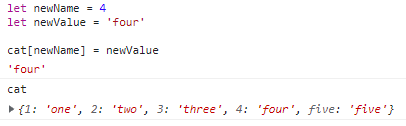
Ладно сначала кое что сам.

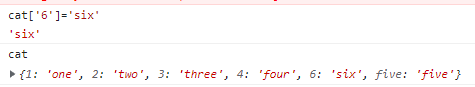






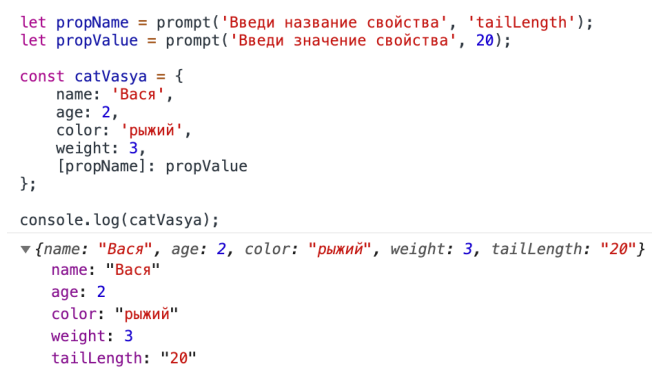
Это один способ.





**Видно независимо от способа имя объекта приходит не в виде строки.**

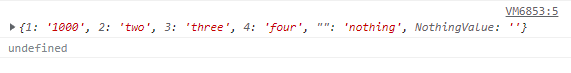
А книга показывает вот такой способ…



Сейчас покажу аналог получше. Тоже через ввод.

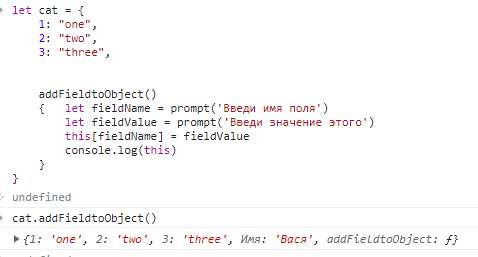


Если же в этой функции в качестве имени задавать уже существующие. То просто будет изменяться существующий объект. **One поменял на 1000**

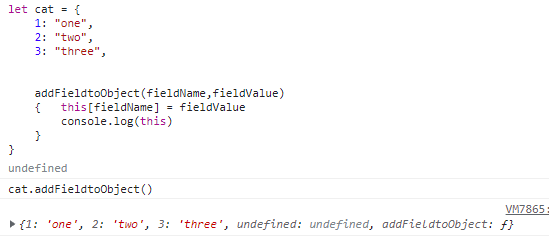


И я тут подумал. А что если эту функцию засунуть в сам объект…

**И да. Я это сделал.** Теперь можно вызвав эту функцию добавить новые поля в объект.



Можно переписать, чтобы поля вводились вне объекта.



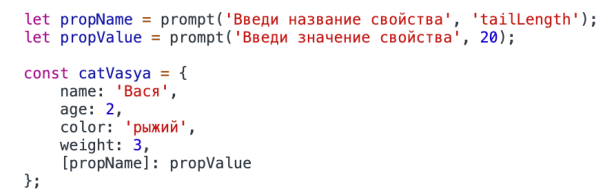
Ничего не прислал получил **undefined: undefined.** Чтобы этого избежать можно значения по умолчанию задать. Кстати вот так добавить поле не вышло. Нужны кавычки.



**Идеально.**

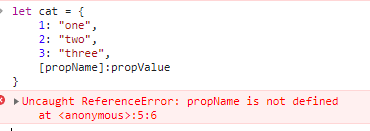
****

**Ладно, теперь можно дальше позволить этой книге меня учить.**

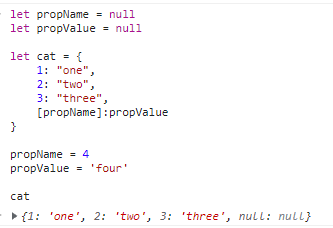


А что если сначала написать объект. А уже потом создать эти переменные?

Ой.. Что? Не определены переменные?

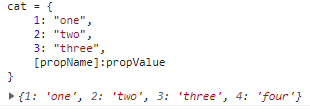


Ладно объявим их пустыми сначала.



Тоже нет? Странно…(сарказм)

И только после этого если еще раз ввести поля, то эти значения обновятся.



**Ладно.. ладно. Книга.**

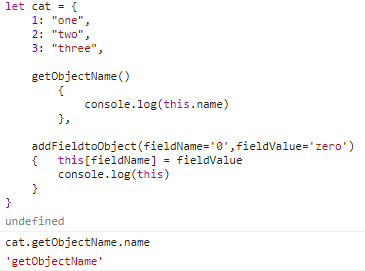
****

Хах. Книга даже не стала дальше объяснять.

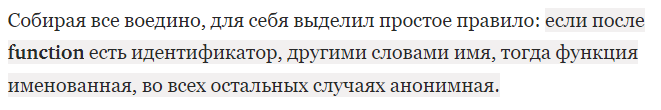
**Именование свойств объекта** это важный аспект, есть правило, которое гласит, что имена свойств должны быть указана как **имена существительные.** То есть ни какие-то там цифры. Ну каюсь. Это я чисто для примера писал.

А вот действия, которые может выполнять объект, как мы уже знает. Это функции. О функция внутри объекта в следующем уроке. Ок. Но блин. Надеюсь я о них чего-то не знаю.

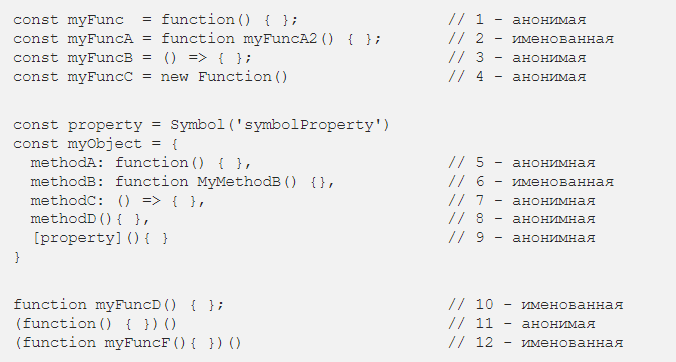
Не знаю зачем мне это, но с помощью метода **name** у объекта можно получить строчное имя введенной функции.

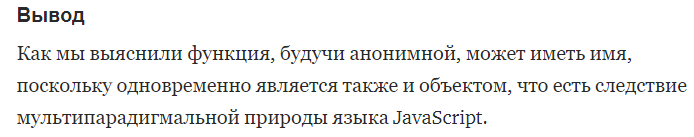


**Небольшое отступление. Что** **есть анонимная функция**.



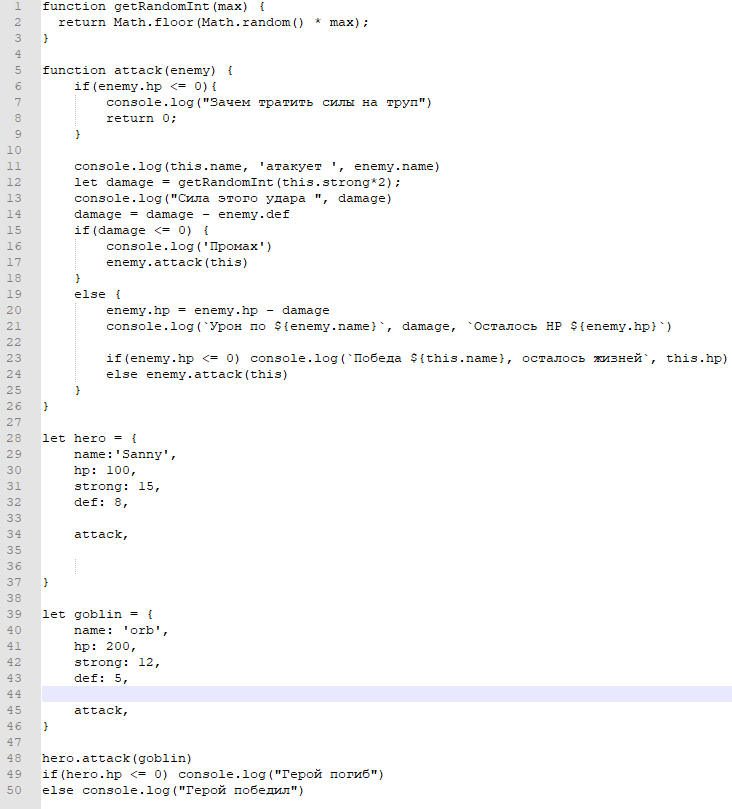
**И любая стрелочная функция анонимная**

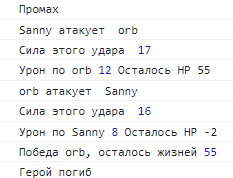
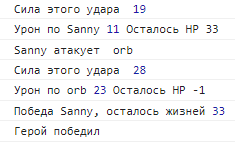
****

****

**Объекты. Методы.**

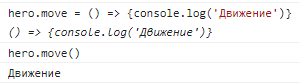
Методы объекта. Это его функции. Они могут делать что-то не определенное, а могут, например взаимодействовать с этими свойствами. **Захотелось попробовать создать сразу два объекта, один из которых будет взаимодействовать с другим.**

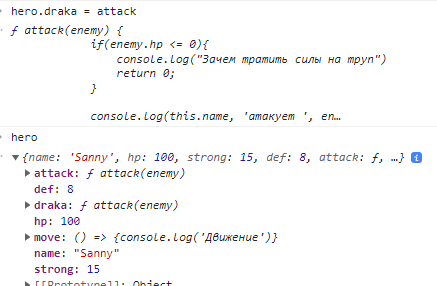
****

** **

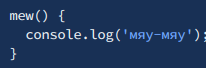
**Не плохо попрактиковался.**

**Методы** добавляются в функцию точно также как и **свойства**

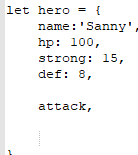




Ну а если писать функцию внутри объекта, то самую быструю запись уже показывали.



Если функция уже объявлена, то можно просто вставить ссылку на нее.





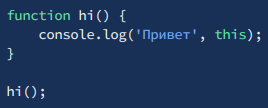
Да, в курсе.

**Именование методов.**

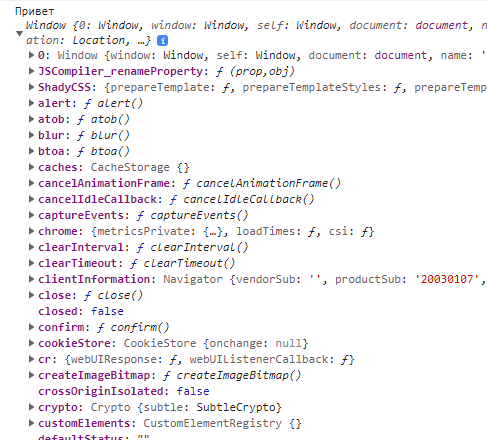
Методы называют как глаголы. Да.. очень полезная инфа.

**This, call, apply, bind**

А вот это интересно. Все эти слова связаны с одним словом. **Контекст.** Говорят это сложная и непонятная тема для новичков. Попробуем создать функцию, которая выводит **this.**

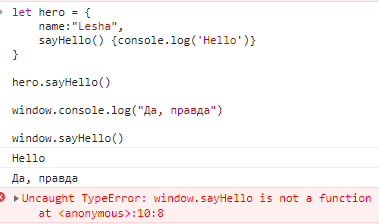
****

Вызвав эту функцию, мы увидим вот такую строку:



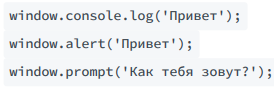
Что это за объект **window.** Начнем с того, что все методы, которые часто вызываются вроде: **console.log, alert, prompt** и т.д. Находятся внутри объекта **window** и являются его методами.

Похоже это правда.

****

**Ну не совсем. Window тут просто не видит метод, потому что он объявлен внутри объекта.**

Но это правда

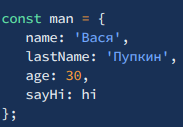


Поэтому, когда мы вызываем функцию любую**,** вызывается она также из объекта **window.**

 == 

Наша функция запускается из объекта **windows.** То есть **контекст,** в котором выполняется эта функция также равен объекту **window.**  А ключевое слово **this** как раз и содержит внутри себя **контекст,** в котором вызывается функция**.**

**Создадим объект.**



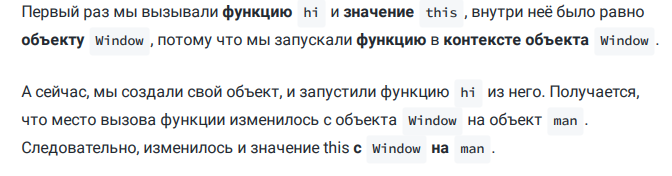
И попробуем воспользоваться методом **sayHi,** который выводит **Привет, this**



**Результатом будет контекст этого объекта.**



Я слегка ошибался, думал что под **this** идет обращение именно к самому объекту. Хотя по сути разнится нет.



Вот чем является **this.**

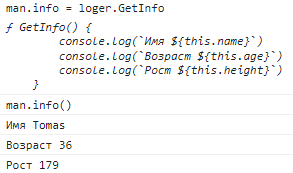
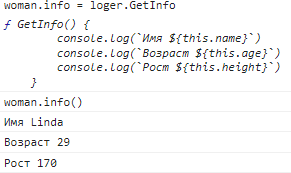
**Значение this равно тому объекту в контексте которого было вызвано.**

А теперь эти страшные методы: **call, apply, bind.**

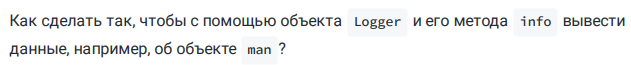
**Метод bind.**

Создал 3 объекта один из которых это содержит метод. Попробую метод из этого объекта присвоить этим двум объектам.



Допустим. Ну так о чем метод **bind?** Очевидно, что если вызвать метод **loger.info** то вызывается он из контекста объекта **loger.** А у **loger** просто нет таких полей, как: **name, age, height.** Поэтому и выводится 3 раза **undefined.**



Ну…

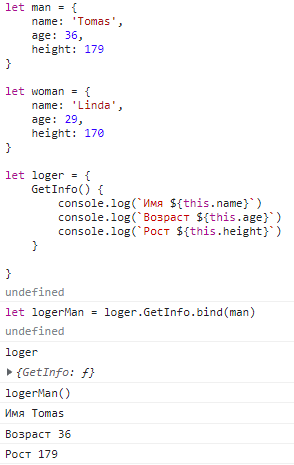
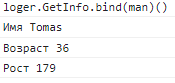
**имя\_объекта.название\_нового\_метода = имя\_объекта\_откуда\_копируется\_метод.имя\_метода**

То есть **man.info = loger.GetInfo**

Но похоже это не единственный способ. И для этого и придуман метод **bind?**

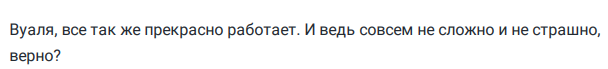
****

Хм… похоже не совсем так. Этот метод не добавляет, метода в **объекты.** Но позволяет привязать к методу другой контекст. В данном случае мы в переменную l**oggerMan** записали ссылку на метод объекта **Logger**, но для контекста **man** с помощью метода **bind.**

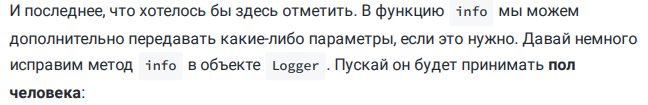
 

Функцию не обязательно сохранять, **можно сразу и вызвать** **↑,** ух а если ее переписать, то можно будет сделать свой ввод контекста.

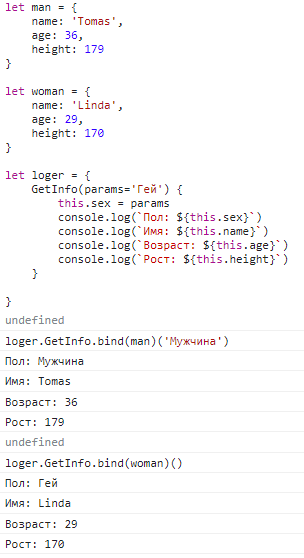
У каждой функции в **JavaScript** существует метод **bind,** с помощью него можно **любой функции** задать контекст, внутри которого она должна будет выполняться. Метод bind возвращает новую функцию, к которой будет привязан указанный **контекст**.



Да, верно. Спасибо за новое знание.

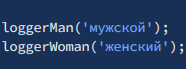


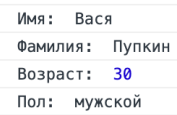
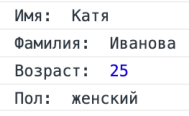
Сейчас попробую, не подглядывая. Так, не совсем понял. Мне нужно добавить новое поле в объект с помощью этого метода? Или в самих объектах в ручную добавить его?



Типо так? Пойду узнаю ответ.



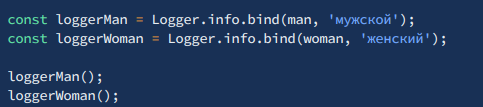
 

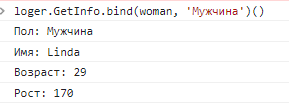
 

… … …

Мдах.. Ладно. В чем моя претензия? В том что, пол не добавляется как свойство. Он просто выводит введенный параметр.

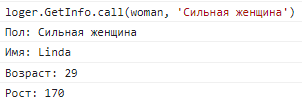
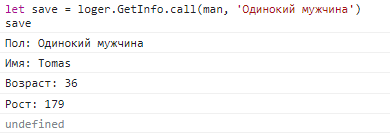
Есть кстати другой способ передачи параметров, с помощью **bind.**



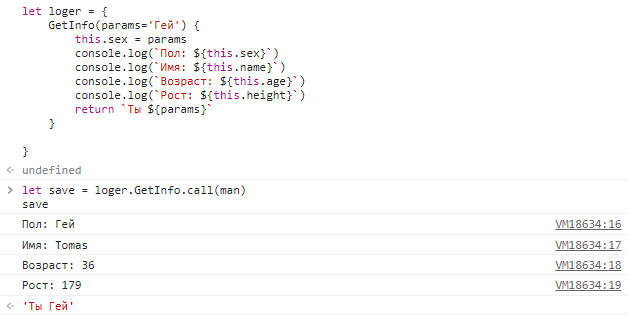


**Первый аргумент в методе bind это всегда контекст. А после можно через запятую указывать сколько угодно параметров.**

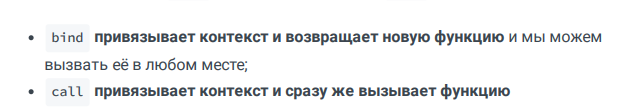
**Метод call** этот метод делает абсолютно тоже, что и **bind.** Только вместо возвращения метода или метода с параметрами, **call** сразу его выполняет. Удобно. Не надо писать две доп. скобки(). Метод **call** ~~ничего не возвращает.~~ Метод возвращает возвращаемый методом результат.

Хм… однако, если метод будет что-то возвращать. То он ведь будет сохраняться?



Да будет.

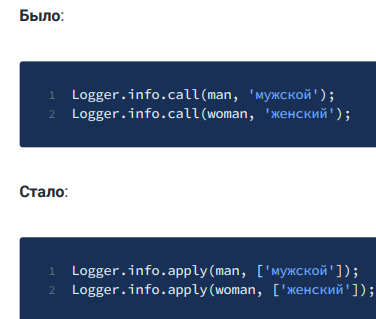




**И CALL ВОЗВРАЩАЕТ ТО, ЧТО ВОЗВРАЩАЕТ ФУНКЦИЯ!!!**

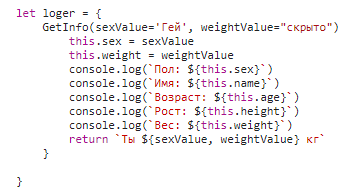
**Метод apply** делает тоже самое, что и **call,** но с **одним отличием.**

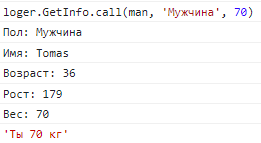
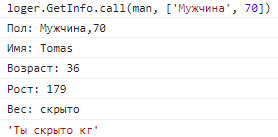
**call** после первого аргумента принимает любое количество параметров через запятую. А метод **apply** после первого аргумента принимает **один** аргумент – **массив параметров.** Этот метод принимает только массив.



Хочу провести пару экспериментов. Напишу функцию, которая принимает 2 параметра.

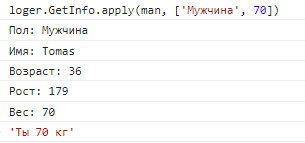
Для метода **call** результат будет таким:

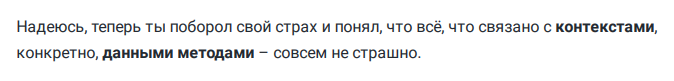


А для метода **apply** таким:

****

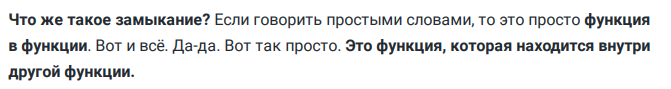
**Именно это я и ожидал.**

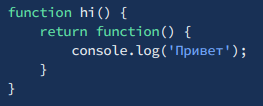


Ну страшно не было. Но могу понять новичков у которых с этими методами были сложности.

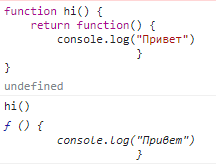
**Замыкания**

Замыкание… Может что-то связанное с рекурсией? Сейчас узнаем.





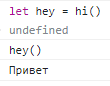
Ну смотрю я этот код. И думаю, что он отработает так. Ну короче. Я не думаю, что выведется привет. Я думаю… Ладно проверю.

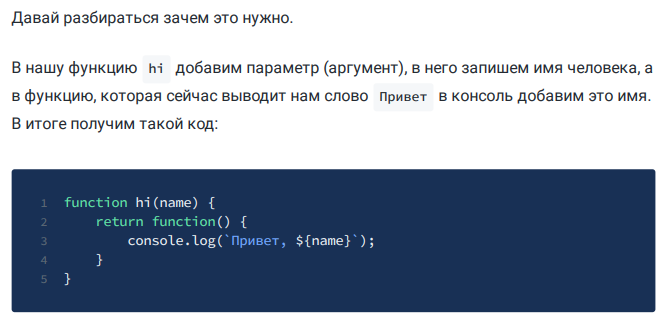


А вот оно. А если написать так?



Результат выполнения функции hi это другая функция. Чтобы уже она отработала, нужно этот результат выполнения вызвать как функцию, например так:

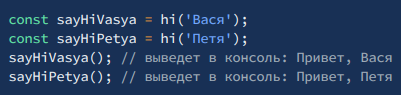






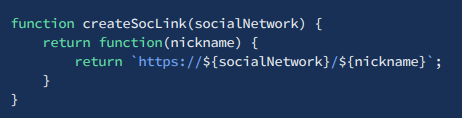
Кажется начинаю понимать. Имя мы передаем только в функцию hi. Но возвращаемая функция имеет то значение, которое мы передали. То есть оно становится часть области видимости? Сейчас узнаем.





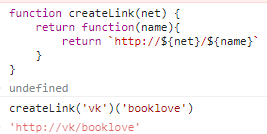
**Создание ссылок для соц. сетей.**

Пока не совсем понятна полезность, поэтому рассматриваем другой пример.

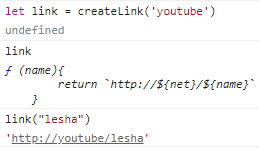


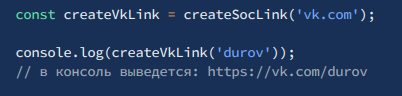
Ну это тоже самое. Только вместо вывода вторая функция будет возвращать значение.

Если я перепишу эту функцию. То вот как я ее вызвать могу.

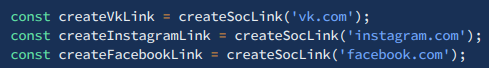


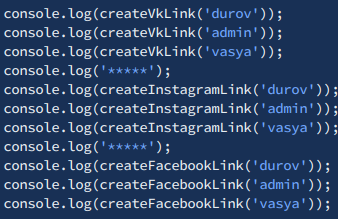
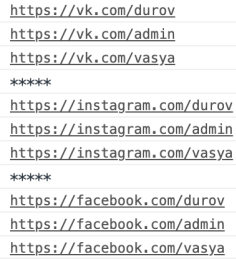




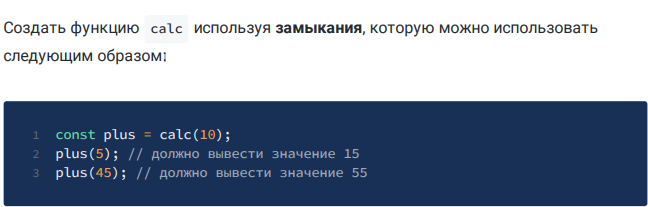


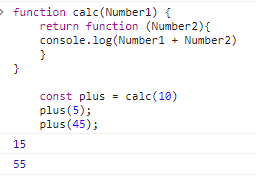


А.. и вот таким образом у нас готовы 3 функции, которые генерируют ссылки на 3 социальные сети. И теперь, вызывая эти функции с разными именами, будут создаваться разные ссылки. Ладно. Но не то чтобы мега полезно.

Ну тут ДЗ я сделаю.



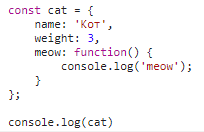
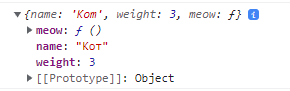


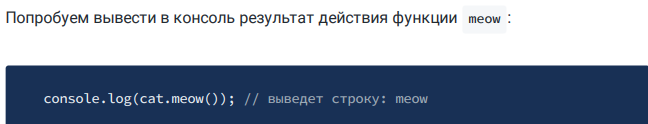
Все.

**Прототипы**

**Прототипы** связаны с **объектами**. Поэтому сразу создаем и выводим **объект.**



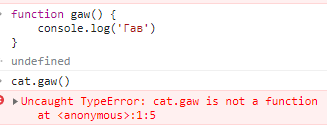
Да. Только зачем ты это еще и в **console.log()** обернул?



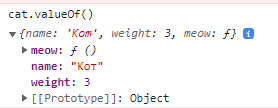
Если попытаться вызвать несуществующий метод. Очевидно, будет ошибка.



Gaw не является функцией, ведь я не определял эту функцию внутри объекта. Даже если такая функция будет существовать:



А что если ввести такие функции?



Этот метод вернул **контекст**



А этот метод вернул преобразованный в строку массив [object Object].

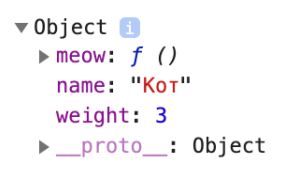
Итак, разберемся. Где же все эти методы хранятся. Спойлер. В такой же штуке как

**[[Prototype]]: Object** – это в хроме. В Mozilla к примеру вот так:

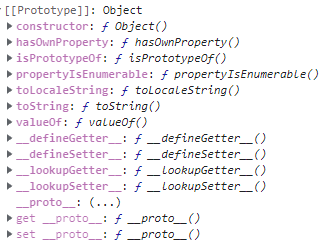


Но суть не меняется.

В учебнике вообще вот так:



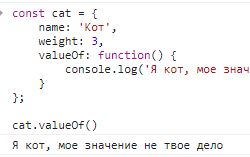
Итак, что же у нас там хранится.



**Ладно.** Вижу как минимум уже знакомые **toString** и **valueOf.**

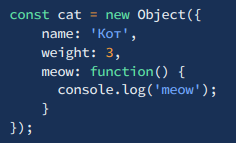
Когда мы вызываем какой-то метод через объект, **JS** сначала пытается найти этот метод внутри тела самого объекта. А когда не находит, уже обращается к **proto** или **Prototype.** Ну это значит, что я в объекта могу назвать функцию также, тем самым заблокировав функцию внутри **Prototype?**

Нука:

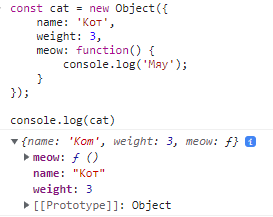
  
**Ага.**

Напомню. Создавая объекта так:

**Let cat = {}**  мы используем упрощенный метод. Второй же:



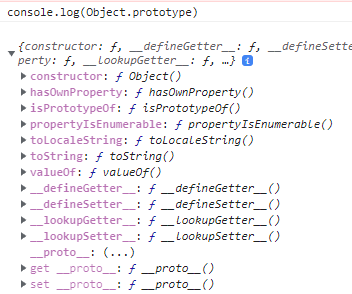
Создаем новый объект, используя ключевое слово **new.** И внутрь этого **Object** передаем наш **объект.** От такого способа ничего не изменится. Все тоже.



Это доказывает, что когда мы используем краткую запись {} все равно создается новый объект через функцию **new Object{}.** Поэтому у любого объекта будут эти свойства.

Все создаваемые в **JS** объекты основываются на базовом классе **Object.**

У класса **Object**  имеется свойство **prototype.**

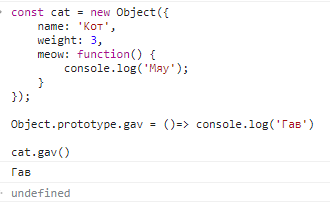


И именно эта штука попадает в каждый созданный **объект.**

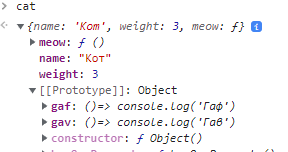
Так для чего это нужно и как это использовать?

Как насчет добавить метод гав не нашему коту, а объекту.

И вот кот загавкал.



А заглянув в объект можно увидеть в **prototype** новые поля.

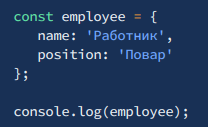


И такие поля будут созданы у всех объектов. Даже тех, что будут созданы позже.



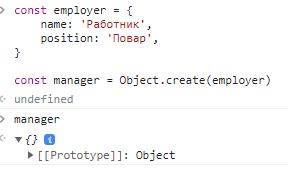
**Object.create - это** метод, который есть у класса **Object.** О нем стоит рассказать.

Чтобы объяснить как он работает понадобятся 2 объекта.



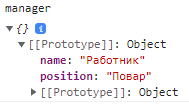
Для создания второго воспользуемся методом create. В него помещаем объект работника.





Ну… разницы я не заметил. Пустой объект. Вроде. Но…

Внутри **Prototype** можно найти нашего рабочего.

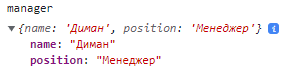


****

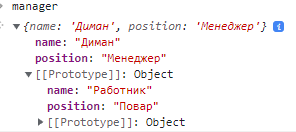
**То есть,** создав **empoyer,** мы создали объект, **прототипом** которого, является **Object.** А создавая **manager,** мы использовали метод **Object.create(),** тем самым создав объект, прототипом которого является тот объект, который был положен в метод, то есть **employer.**

Ну допустим добавим менеджеру имя и должность.





А наш прошлый работник? Он на месте.



Но теперь если пытаться обратится к полю **manager.name,** то будем получать имена и позиции уже менеджера.



А чтобы обратится к работнику, придется заглянуть в **прототип.**

****

**Интересно можно ли без? \_\_ \_\_**

Если подытожить, то **прототип** – это **базовый объект другого объекта.** Этот базовый объект присутствует у других объектов и **расширяет** их возможности

**Кругом обман.**

Все прошлые материалы о типах данных немного недоговаривали.

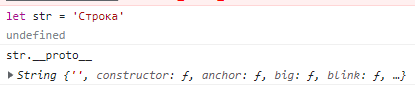
К примеру создаем обычную строку.



Но у нее есть методы. Хм... А откуда?



Этот метод ведь не определяли. Но прототип у нее есть.



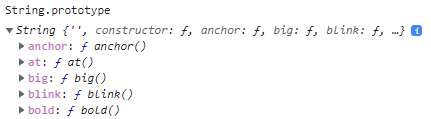
Дело в том, что строковый тип тоже объект, а именно **String.** А **String** уже основан на объекте **Object.**

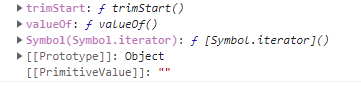
Поэтому когда мы создаем строку.



Это равносильно этому:







**Все в JavaScript создается на основе класса Object. Он главный в JS.**

**Promises (Обещания)**

Напомню, **JS** синхронный язык. То есть код выполняется строчка за строчкой **последовательно**.

Ладно. Но в языке быстро понадобилась асинхронность. Зачем? Ради скорости. Социальные сети и т.д. так быстро загружаются благодаря **асинхронным операциям.**

Когда заходишь в соц. сеть браузер сразу отсылает множество запросов, чтобы получить какие-то данные. Затем он из них формирует страницу, которую пользователь видит. На все эти запросы нужно время. И если бы они выполнялись по очереди, то пришлось бы очень долго ждать.

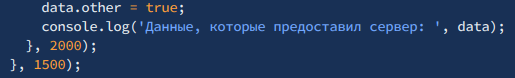
Благодаря **асинхронности,** все запросы отсылаются одновременно, и весь остальной код не ждет ответ от них, а продолжает выполняться. А когда **асинхронная операция** заканчивает свое исполнение – отрабатывает какая-либо заранее подготовленная функция.

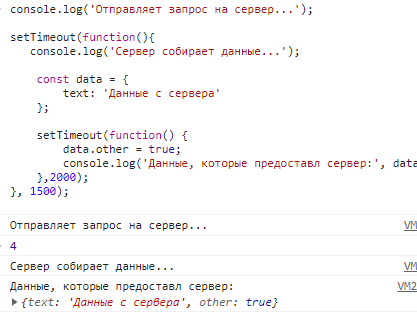
Асинхронные операции уже встречались: **setInterval, setTimeout**

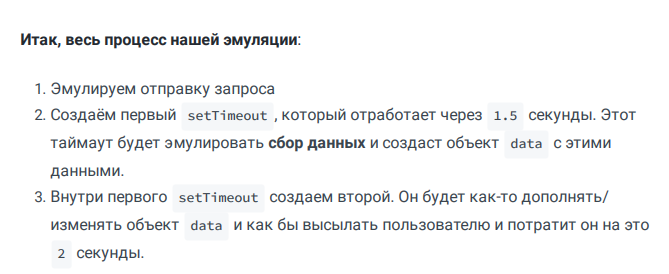
**Эмуляция работы с сервером с помощью setTimeout.**

Да, с помощью этой функции можно сделать вид, что отправил запрос, и затем получил ответ и выполнил.

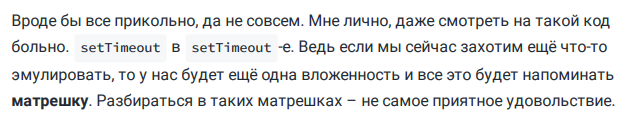
****

****

****

****

Вот все это отработало и ушло на это **3.5** секунды.



**Полностью солидарен.** Поэтому перепишем эту функцию с помощью **Promise.**

**Эмуляция работы с сервером с помощью Promise**

Сначала создадим пустой **Promise.**

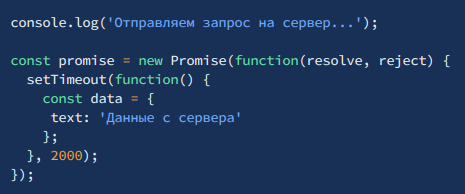
****

Как можно заметить это тоже **Класс,** основанный на **Object.**

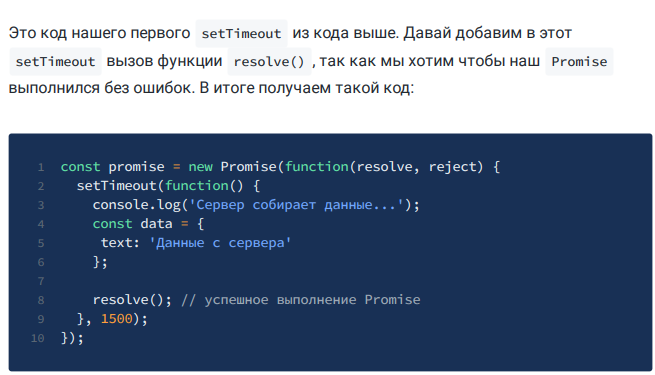
В конструктор данного класса передается только **один** аргумент. Это **callback-функция** с **двумя** аргументами**.**



А эти аргументы на самом деле являются функциями. Благодаря ним и будет осуществляться контроль выполнения **Promise.** Например, внутри обещания будут какие-то проверки и в зависимости от их результата, будет вызываться либо функция **resolve**, либо **reject.**

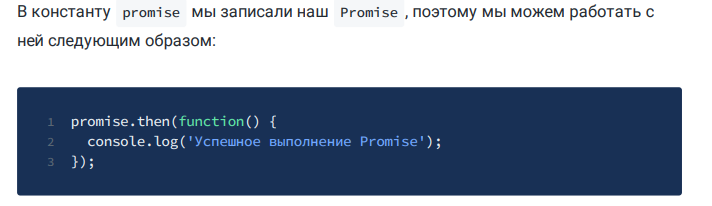


Итак внутри **callback-функции promise** записали вызов нашего **SetTimeout**

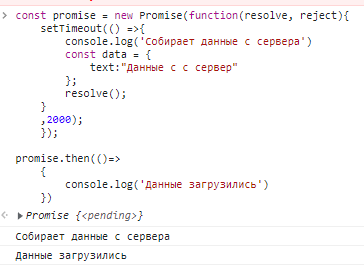


Почему ты добавил **resolve()** я понял. А почему у тебя это уже другой запрос блин, 1500 секунд. У тебя оба одинаковых **promise?** А если ты прошлый код переписал… Ааа башка кипит.

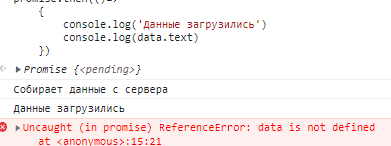
Стр 206. Не могу.

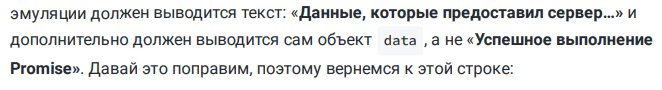


Попробую.



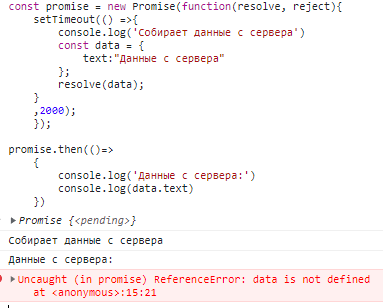
Однако в методе **.then** мне не удалось обратится к данным из объекта.



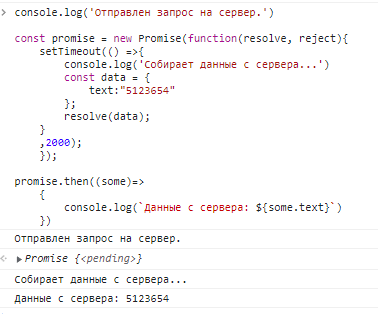


Да, у меня пока не вышло. Так как надо? Проблема в том, что в этой **callback-функции** нет никакого объекта **data.** Чтобы эту проблему решить, нужно в метод **resolve** эти данные передать.

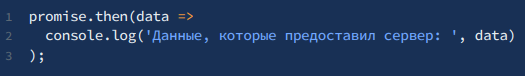
Попытка номер 2. **Не удачно**

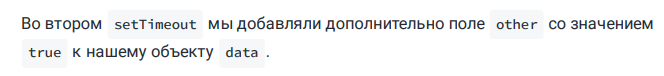


Тут моя ошибка в том, что я отправил данные в функцию **resolve.** Но в самой функции ничего не принял. Вот решение.

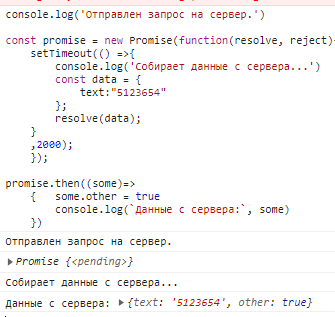


Или вообще вот так написать надо было.



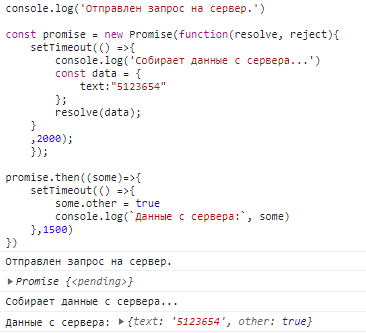


Ладно... сейчас



Вот. Ах туда нужно еще задержку в 2 секунды поставить.

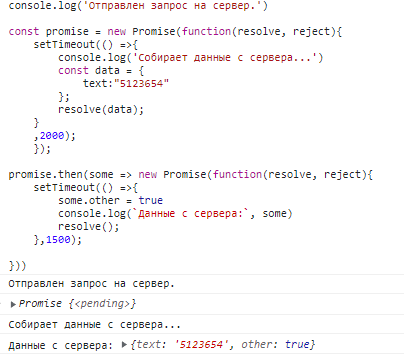
Вот.. Вроде со всеми условиями.



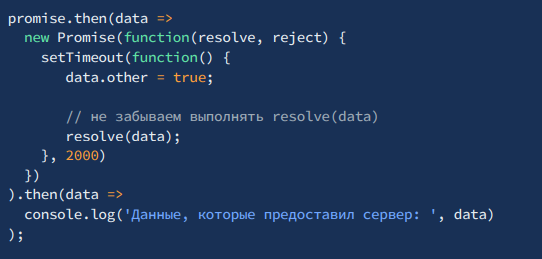
Казалось бы, все круто. Но не совсем. Дело в то, что второй SetTimeout тоже должен иметь свой promise. Нужно, чтобы callback функция из **then** возвращала новый promise



Только я не понял. Зачем мы второй раз в resolve отправляем data. У меня из без этого все отработало.



Ааа… крч. Во втором promise будет изменение. А вывод уже нужно реализовать в третьем, полагаю. Только обращение к этому Promise очень странно выглядит.

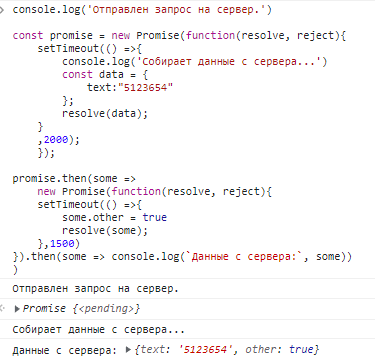


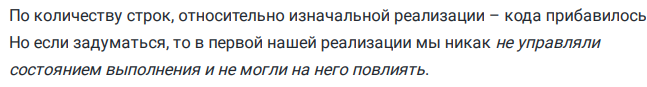
Попробую хоть как переменную внутри этого promise его объявить. **Не вышло.**

Вот тоже самое и если честно. Такой бардак. Типо он выше писал…

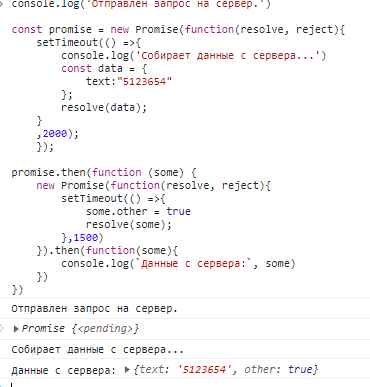


А вот этот код как будто на много удобнее…





Ну тут правда. Но как же не читабельно. Может станет понятнее если заменить стрелочные функции на обычные…

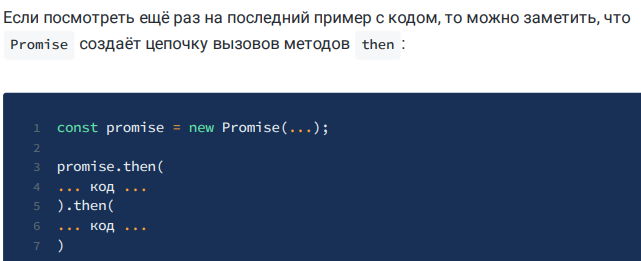


Не все также. Сложность именно в том, чтобы уследить где заканчивается одна и начинается другая функция, чтобы правильно закрыть скобки.

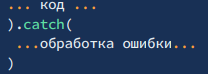
**Метод reject** служит для того, чтобы сообщить, что **Promise завершится.**

Обработчик для **resolve - .then**

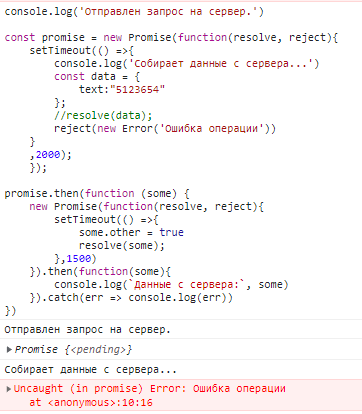
Обработчик для **reject - .catch**



Место для **catch** в этом списке в самом конце.



Только **reject** должен принимать переменную типа **Error** созданную с помощью **new Error()**



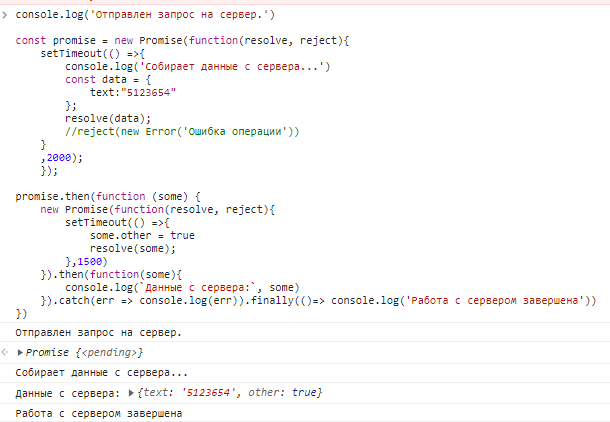
Видно что передалась **ошибка операции.**

Кроме того, кроме методов **then** и **catch** существует метод **finally.**

Он исполняется всегда, независимо от того вызвался **reject** или **resolve.**

Он размещается в самом конце.



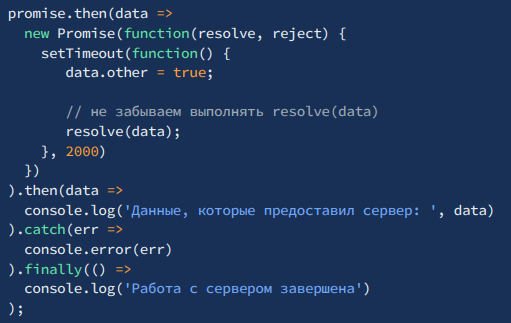
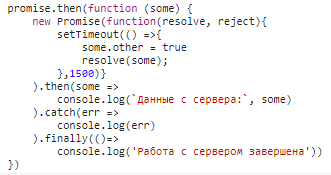




**А у меня тогда скажи что?!**

Попробу переписать опять без стрелочных

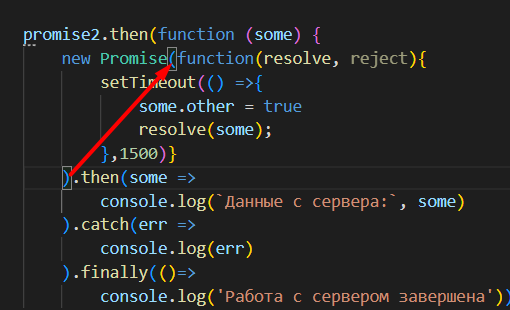
**Серьезно. В чем разница:**



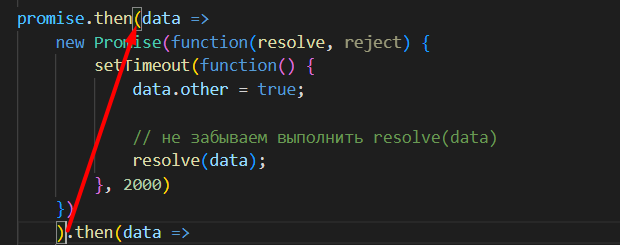
**А если переписать 1/1**



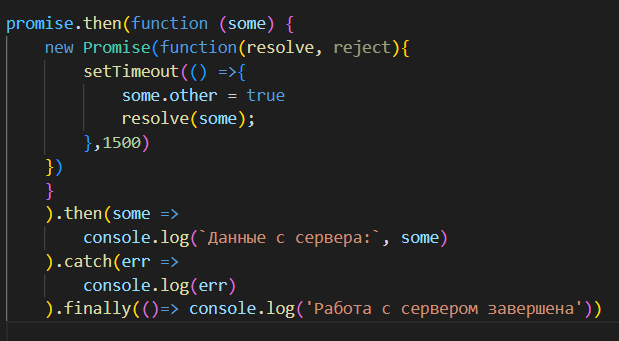
**Да. Так работает. Это я написал, что-то не верно. Но что?**



В моем первом коде, первый **then** обращался к **new promise,** а должен был…



Вот так старый код работает.



Хотя и выглядит уродски.

**Конец книги.**